University of Alberta

Improving Rich Internet Applications through Software Refactoring

by

Ming Ying

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering

© Ming Ying
Spring 2012
Edmonton, Alberta

Dedicated to my beloved parents …

獻給我最愛的父母

Abstract

With the advent of Rich Internet Application (RIA) technologies which are crucial to Web 2.0 sites, Internet user experience has moved from the click-and-wait mode to a richer, faster and more interactive mode. Instead of refreshing the entire web page every time when a user requests a change, only updated information within the web page is modified. This allows RIAs to behave and feel more like desktop applications.

Due to the evolving nature of RIAs, many efficiency issues need to be resolved before RIAs can behave like desktop applications. Ensuring the efficiency of RIAs is now an important issue. This is the reason why many web browsers advertise the speed of their JavaScript engines as one of the key features. Additionally, web application performance issues can affect corporate revenues because with every 1-second delay, customer satisfaction decreases.

Two of the most popular RIA technologies are Adobe Flash and Ajax, and the efficiency of RIAs using both of these technologies can be improved. This dissertation introduces refactoring as a method to improve the efficiency of applications built using these platforms. Programmers using the techniques and tools introduced in this dissertation can greatly improve the efficiency and user experience of their applications. More specifically, the thesis introduces four techniques and tools.

- A refactoring tool called ActionScript Refactoring Tool (ART) is introduced to improve the efficiency of Flash applications by rewriting ActionScript 3.0 code.
- To aid programmers embed Flash programs effectively, a refactoring tool called FlashembedRT is introduced. This tool can refactor five popular markup-based Flash embedding methods to a JavaScript-based Flash embedding method called flashembed.
- A refactoring approach to aid programmers transform their XML data structures into JavaScript Object Notation (JSON)-based structures to improve the efficiency of their applications is presented. A proof of concept tool called XtoJ shows that this transformation can be automated to help programmers rapidly access the efficiency gained when JSON is used.
- A refactoring system called Form Transformation Tool (FTT) is proposed as a technique to help programmers convert traditional web forms into Web 2.0 Ajax-enabled forms.

Acknowledgement

First and foremost I offer my sincerest gratitude to my supervisor, Dr. James Miller, for his guidance, encouragement and support during my research and study at the University of Alberta. Without him this dissertation would not be possible. Dr. James Miller is not only the mentor for my research, he is also my mentor for life here in Canada. I did learn a lot from him. Thank you James!

I also would like to thank my entire family in China for their continual support. Although they are not here with me, they still provided encouragements and support to help me get through difficult time periods.

Thank you Toan Huynh for providing valuable advice, helping me solve problems and editing this dissertation.

My friends and colleagues at the University were also a great source of information and kept my life more enjoyable during the compilation of this dissertation, so I would like to extend a thank you to all of them.

Table of Contents

Chapter 1 Introduction 1
- 1.1 Chapter 2 - Background 3
- 1.2 Chapter 3 - Related Works 3
- 1.3 Chapter 4 - Refactoring ActionScript for Improving Application Execution Time 3
- 1.4 Chapter 5 - Refactoring Flash Embedding Methods 3
- 1.5 Chapter 6 - Refactoring to Switch the Data Exchange Language for Improving Ajax Application Performance 4
- 1.6 Chapter 7 - Refactoring Traditional Forms into Ajax-enabled Forms 4
- 1.7 Chapters 8 - Conclusion and Future Work 5

Chapter 2 Background 6
- 2.1 Refactoring 6
- 2.2 The Flash Execution Model 8
- 2.3 Ajax 10
- 2.4 XML 11
- 2.5 JSON 11

Chapter 3 Related Work 13
- 3.1 Related Work on Refactoring 13
- 3.2 Related Work on Speeding up Web Applications 15
- 3.3 Related Work Specific to Chapter 4 16
  - 3.3.1 Dynamic Optimization 16
  - 3.3.2 Compiler Optimization 17
  - 3.3.3 Speeding up Embedded System 19
  - 3.3.4 Speeding up Distributed System 20
  - 3.3.5 Summary 21
- 3.4 Related Work Specific to Chapter 6 21
- 3.5 Related Work Specific to Chapter 7 22

Chapter 4 Refactoring ActionScript for Improving Application Execution Time 23
- 4.1 Motivating Example 24
- 4.2 Three Strategies for Speeding up Flash Applications 25
  - 4.2.1 Optimizing Bytecode Directly 25
  - 4.2.2 Performing Optimizations on JIT 25
  - 4.2.3 Refactoring 25
- 4.3 Design of ART 26
  - 4.3.1 Characteristics of the System 26
  - 4.3.2 Maintainability 27
  - 4.3.3 ART's Execution Model 29
  - 4.3.4 Overview of Our Refactoring Cycle 30
  - 4.3.5 Bad Smells and Refactoring Solutions in ActionScript 3.0 31
- 4.4 Evaluation 34
  - 4.4.1 Methodology 34
  - 4.4.2 Testing the Performance of Refactoring Patterns 34
  - 4.4.3 Testing the Performance of Real Applications 41

Chapter 5 Refactoring Flash Embedding Methods 45
- 5.1 Markup-based Flash Embedding Methods 47
  - 5.1.1 The <embed> tag 47
  - 5.1.2 The <object> tag 48
  - 5.1.3 Twice Cooked 49
  - 5.1.4 Nested Objects 50
  - 5.1.5 Flash Satay 51
- 5.2 JavaScript-based Flash Embedding Methods 51
- 5.3 Refactoring Flash Embedding Methods 52
  - 5.3.1 Bad Smell Detection 53

5.3.2 HTML Code Rewriting 54
5.3.3 Flashembed Code Generation 54
5.3.4 Grammar for Transformation 55
5.3.5 System in Operation 56
Chapter 6 Refactoring to Switch the Data Exchange Language for Improving Ajax Application Performance 58
6.1 Refactoring XML into JSON in Ajax Applications 59
6.1.1 Example 59
6.1.2 Performance Comparison 60
6.1.3 Methodology 62
6.2 System Components 62
6.2.1 XML to JSON Converter 63
6.2.2 JavaScript Code Transformer 70
6.2.3 JavaScript Code Generator 74
6.3 Evaluation 79
6.3.1 Methodology 79
6.3.2 Testing Results for Ten Ajax Applications 79
6.3.3 Variables Influencing the Response Time 82
Chapter 7 Refactoring Traditional Forms into Ajax-enabled Forms 85
7.1 Motivating Example 85
7.1.1 Problem 1: Submission 85
7.1.2 Problem 2: Validation 87
7.2 Methodology 87
7.3 Refactoring Traditional Forms into Ajax-enabled Forms 88
7.3.1 Form Submission Transformer 88
7.3.2 Validation Code Generator 91
7.3.3 QUnit Code Generator 94
7.4 Transforming the Example Form into an Ajax Form 95
7.4.1 Record the Form’s Submission Process 95
7.4.2 Add Test Case 96
7.4.3 Transform the Registration Form into an Ajax Form 96
7.4.4 Add Validations 98
7.4.5 Replay the Form’s Submission Process and Execute the Unit Test 100
7.4.6 Case Study 2 101
Chapter 8 Conclusion 105
8.1 Future Work 106

List of Tables

Table 4.1 Testing Results for Different Patterns in Different Configurations 36
Table 4.2 Testing Results of Refactoring Patterns for Flash Applications 43
Table 5.1 An Example Configuration 47
Table 6.1 Results for Different Scenarios 61
Table 6.2 The XML Documents Size for the Ten Tested Applications 80
Table 6.3 The Size of Some Large XML Documents 81
Table 6.4 Testing Results for the Ten Ajax Applications 81
Table 6.5 Testing Results for Different Number of Objects 83
Table 6.6 Testing Results for Accessing Different Objects in Different Structures 84
Table 6.7 Testing Results for Accessing a Node with Different Lengths 84
Table 7.1 The Number of Lines of Code Generated by FTT 103
Table 7.2 Testing Results for the Response Time before and after Refactoring 104

List of Figures

Figure 2.1 Refactoring Process with a Tool 6
Figure 2.2 Client Side's Flash Execution Model 8
Figure 2.3 The Classic Web Application Model 10
Figure 2.4 The Ajax Web Application Model 10
Figure 2.5 An Example of XML Data Structure 11
Figure 2.6 An Example of JSON Data Structure 11
Figure 4.1 The Interface of the Tetris Game 24
Figure 4.2 One Option for ART's Execution Model 29
Figure 4.3 ART's Execution Model 30
Figure 4.4 Refactoring Cycle 31
Figure 4.5 The Template of the Refactoring Patterns 32
Figure 4.6 Improvement of Refactoring Patterns 42
Figure 4.7 The Effects of Code Structure on Pattern 44
Figure 5.1 The Combined Ajax and Flash Working Model 45
Figure 6.1 The Ajax RSS Reader Web Page 59
Figure 6.2 JavaScript Code for XML-based Ajax RSS Reader 60
Figure 6.3 JavaScript Code for Accessing the JSON File 74
Figure 6.4 The Generated JavaScript Code for Accessing the RSS JSON File 78
Figure 7.1 The User Registration Web Page of JspCart 86
Figure 7.2 The Validation Selection Window 93
Figure 7.3 The Output of the Form Submission before Refactoring 96
Figure 7.4 The Registration Form after Refactoring 99
Figure 7.5 The Output of the Form Submission after Refactoring 100
Figure 7.6 The QUnit Testing Result 101
Figure 7.7 The Product Management Page after Refactoring 102

List of Acronyms

A

ABC: ActionScript bytecode
Ajax: Asynchronous JavaScript and XML
ANTLR: Another Tool for Language Recognition
API: Application Programming Interface
ART: ActionScript Refactoring Tool
AS2: ActionScript 2.0
AS3: ActionScript 3.0
AST: Abstract Syntax Tree
AVM: ActionScript Virtual Machine

C

CDN: Content Distribution Network
CPU: Central Processing Unit
CS3: Adobe Flash CS3 Professional
CS4: Adobe Flash CS4 Professional
CSE: Common Sub-expression Elimination
CSS: Cascading Style Sheets

D

DCE: Dead Code Elimination
DMC: Direct Mapped Cache
DOM: Document Object Model

E

EMS: Experience Management System
EBNF: Extended Backus–Naur Form

F

FTT: Form Transformation Tool

H

HTML: Hypertext Markup Language

I

IPO: Interprocedure Optimization

J

JIT: Just-in-Time
JSON: JavaScript Object Notation
JSP: Java Server Pages

K

KISS: Keep it simple, stupid

L

LLVM: Low Level Virtual Machine

M

MD: Machine Code
MIR: Macromedia Intermediate Representation
ML: Metalanguage
MP3: MPEG Layer III

P

PGO: Profile-Guided Optimization
PS: Publish Settings

R

RIA: Rich Internet Applications
RMD: Relative Mean Difference
RSS: Really Simple Syndication

S
SGML: Standard Generalized Markup Language
SDT: Software Dynamic Translator

U

URL: Uniform Resource Locator

V

VM: Virtual Machine

W

W3C: World Wide Web Consortium

X

XHTML: eXtensible HyperText Markup Language
XML: Extensible Markup Language

Chapter 1 Introduction

The current rise in popularity of Web 2.0 applications has significantly changed the way users interact with the web applications. The nature of Web 2.0 requires the growth of RIA technologies (Driver et al., 2005) such as Flash and Ajax. RIA technologies allow richer, faster and more interactive experiences. It breaks the old click-and-wait user experience mode. Instead, by only changing updated information without refreshing the entire page, RIA makes web applications feel more like a desktop application (Hewlett-Packard Company, 2007).

With the advent of the RIAs, users have higher expectations for web applications. The main difference between RIAs and traditional web applications is the level of interaction. The interactions of traditional web applications are restricted to operations on visual objects, such as document, frame and button objects. RIA technologies lay emphasis on a rich and engaging user experience. Graphics, animations and different visual effects are used to create highly dynamic, interactive web pages. Speed becomes an important requirement for the new generation web applications, as it directly affects users' satisfaction.

One of the key requirements for making web applications feel more like a desktop application is performance. That is, the application has to be responsive. Card et al. (1991) demonstrate that for a system "to be seen by a user as responsive", it has to have a respond rate of 0.1 seconds or less. The users will perceive a delay if the system takes one second to respond; and after ten seconds of waiting, they will abandon the task and move on. Traditional web applications are not very "responsive". The interaction model involves the user clicking on something, then waiting. The user first has to wait for the server to process the request. Then the user waits for the server to send the results back to the browser. Finally, the user has to wait for the browser to render the results and display it. RIAs resolve this response-time issue through various technologies.

However, because RIAs are still maturing, many efficiency issues still need to be resolved before RIAs can truly feel like desktop applications. In fact, efficiency is so important that one of the main advantages often advertised for web browsers is the speed of their JavaScript engines. Each time a new version is released, a significant improvement in the JavaScript engine speed can be seen. For example, Microsoft Internet Explorer 9's JavaScript engine is a significant improvement over IE 8[1]. Firefox 9 will have a 44% in improvement speed over the previous version[2]. Efficiency will continue to be an important part of RIAs as they become more common and slowly replace desktop applications. For example, Google Chrome OS is an operating system that has all its applications as RIAs. A report from Aberdeen Group[3] further shows how important performance is for web

[1] http://ie.microsoft.com/testdrive/benchmarks/sunspider/default.html

[2] http://www.tomsguide.com/us/firefox-9-type-interference-support-javascript-compiler-improvement-benchmark,news-12366.html

[3] http://www.aberdeen.com/Aberdeen-Library/5136/RA-performance-web-application.aspx

applications. They estimate that web application performance issues can affect up to 9% of corporate revenues and that customer satisfaction decreases 16% for every 1-second delay. Additionally, WebPerformanceToday.com[4] performed an analysis to show that Macys.com may lose 30% of their revenue due to poor performance. Although newer browsers will continue to have improvements in efficiency, web programmers can also optimize their RIAs to obtain even more gain in efficiency.

RIAs are currently dominated by two technologies: Adobe Flash and Ajax (Asynchronous JavaScript and XML). Adobe Flash is a multimedia platform for creating interactive and animated web sites. Flash movies and games are commonly integrated into web pages as components for entertainment or advertisement. For example, there are many Flash games in Facebook[5]. Flash contains ActionScript 3.0 (AS3) which is an object-oriented scripting language based upon ECMAScript. To view these Flash movies and to execute these ActionScript files, browsers require the Adobe Flash Player add-on. Adobe[6] claims that about 99% of Internet-enabled desktops have Adobe Flash Player installed.

The other popular RIA technology is Ajax. Ajax is comprised of five different technologies (Garrett, 2005):

1. Extensible Hypertext Markup Language (XHTML) and Cascading Style Sheets (CSS) as the presentation layer.
2. Document Object Model (DOM) trees being used for dynamic display and interaction.
3. Extensible Markup Language (XML) (or any other data interchange standards) for data interchange and manipulation.
4. XMLHttpRequest to asynchronously retrieve data.
5. JavaScript binding everything together.

Examples of Ajax applications include: Google Maps[7], Gmail[8], Google Suggest[9], and Facebook. Ajax applications are designed to be more responsive than traditional web application. Hence, the perceived waiting time for users is significantly reduced.

Although Flash and Ajax are two popular technologies for implementing RIAs, the efficiency of these applications can still be improved. This dissertation proposes refactoring as a technique to improve existing web applications through different strategies. The outline and contributions of this dissertation are discussed in the next sections.

4 http://www.webperformancetoday.com/2010/11/30/downtime-versus-slow-page-speed/
5 http://www.facebook.com/
6 http://www.adobe.com/products/player_census/flashplayer/
7 http://maps.google.com/maps
8 https://mail.google.com/mail/help/intl/en/about.html
9 http://www.google.com/

1.1 Chapter 2 - Background

This chapter provides background information on the technologies and systems utilized in the dissertation. More specifically, Refactoring, Flash, Ajax, XML and JSON will be discussed in this chapter.

1.2 Chapter 3 - Related Works

Chapter 3 contains the related works for Chapters 4-7.

1.3 Chapter 4 - Refactoring ActionScript for Improving Application Execution Time

This chapter explores a method to improve the performance of Flash applications because it, especially in mobile devices, directly influences the user's experience. In fact, speed is one of the most important requirements for mobile devices' users (Buyukozkan, 2009). However, Flash programmers usually specialize in graphic design rather than programming. In addition, the tight schedule of projects often makes Flash programmers ignore non-functional characteristics such as the efficiency of their systems; yet, to enhance Flash's user experience, writing efficient ActionScript code is a key requirement. Therefore, Flash programmers require automated support to assist with this key requirement. This chapter presents a "refactoring for efficiency" Flash support system, ART, to help AS3 programmers produce more efficient code by semi-automatically transforming their ActionScript code.

This chapter makes the following contributions.

1. It introduces 43 refactoring patterns. Each pattern contains a bad smell and a corresponding refactoring solution. The performance testing results demonstrate that the refactoring patterns have the ability to make AS3 code faster.
2. A refactoring tool, ART, is produced to improve the efficiency of Flash applications by semi-automatically rewriting AS3 code.
3. It empirically demonstrates that ART significantly improves the efficiency of Flash applications.

1.4 Chapter 5 - Refactoring Flash Embedding Methods

As a first step towards integration of Flash and Ajax technologies, this chapter presents a tool to aid programmers embed Flash into web pages. Two methods of embedding Flash can be used: markup-based Flash embedding methods and JavaScript-based Flash embedding methods. However, the drawbacks of markup-based Flash embedding methods make JavaScript-based Flash embedding methods a better solution. This chapter's contribution is a refactoring tool, called FlashembedRT, to assist programmers with the refactoring of markup-based Flash embedding methods into a JavaScript-based method. More specifically, the tool can refactor five popular markup-based Flash embedding methods to a JavaScript-based Flash embedding method using flashembed[10].

[10] http://flowplayer.org/tools/toolbox/flashembed.html

1.5 Chapter 6 - Refactoring to Switch the Data Exchange Language for Improving Ajax Application Performance

To achieve a more responsive user experience, data transmission rates and performance (on both the client and server) characteristics for Ajax applications are quite crucial. XML and JSON are two popular data exchange formats used by web applications. XML has numerous benefits including human-readable structures and self-describing data. However, JSON provides significant performance gains over XML due to its lightweight nature and native support for JavaScript. This is especially important for RIAs. Therefore, it is necessary to change the data format from XML to JSON for efficiency purposes. This chapter presents a refactoring system (XtoJ) to safely assist programmers migrate existing Ajax-based applications utilizing XML into functionally equivalent Ajax-based applications utilizing JSON. An empirical study demonstrates that this transformation system significantly improves the efficiency of Ajax applications.

This chapter makes the following contributions.

1. It introduces a refactoring approach to convert XML-based Ajax applications into JSON-based Ajax applications. This approach provides programmers with a structured method to transform their Ajax applications.
2. A proof of concept tool called XtoJ is produced to demonstrate that the transformation can be automated. XtoJ includes an XML to JSON Converter, a JavaScript Code Transformer and a JavaScript Code Generator.
3. It empirically demonstrates that JSON-based Ajax applications are more efficient than XML-based Ajax applications; and that programmers can use the introduced method to rapidly access this efficiency gain.

1.6 Chapter 7 - Refactoring Traditional Forms into Ajax-enabled Forms

This chapter explores traditional web forms and how programmers can transform these traditional forms into Ajax-enabled forms. Forms are a common part of web applications. They are used as part of the interaction between the user and the web application. However, forms in traditional applications require entire web pages to be refreshed every time they are submitted. This model is inefficient and should be replaced with Ajax-enabled forms. This chapter presents a refactoring system called FTT to assist web programmers refactor traditional forms into Ajax-enabled forms while ensuring that functionality before and after refactoring is preserved.

This chapter makes the following contributions.

1. A method is introduced to refactor traditional forms into Ajax-enabled forms.
2. It produces a proof of concept tool named FTT to demonstrate that the transformation can be implemented using an automated approach. The aim

of the automated approach is to save developers effort and time while guaranteeing a defect-free transformation.

3. It provides a demonstration showing successful transformations of two applications, one of which is commercially available. Thus, programmers can follow the same process to successfully accomplish the transformation in their own applications.

1.7 Chapters 8 - Conclusion and Future Work

Chapter 8 summarizes the contributions of this dissertation and discusses future work related to the topic of refactoring and efficiency.

Chapter 2 Background

This chapter introduces background information on the technologies and systems utilized in the dissertation.

2.1 Refactoring

Refactoring is a process of restructuring the code without changing its behavior to improve code quality (Opdyke, 1992; Fowler, 1999; Murphy-Hill, 2007). A small behavior preserving change to the source code is made with each refactoring. A significant structural change to the code can be seen once a sequence of these refactorings is applied. The benefits of traditional refactoring include (Fowler, 1999).

1. Refactoring improves the design of software.
2. Refactoring makes code easier to understand.
3. Refactoring helps to fix more bugs.
4. Refactoring saves programmer's time.

One side effect of refactoring is the increase in code complexity and size. The other side effect is "refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning" (Fowler, 1999; Demeyer, 2002).

Refactoring is very popular with programmers because the programming process can be divided into two steps with the technique: (1) write the code to meet the functional requirements. (2) Modify the written code to improve the quality of the code. Kataoka et al. (2002) describe the manual refactoring process and Murphy (2007) describes the process of refactoring with a tool, as shown in Figure 2.1.

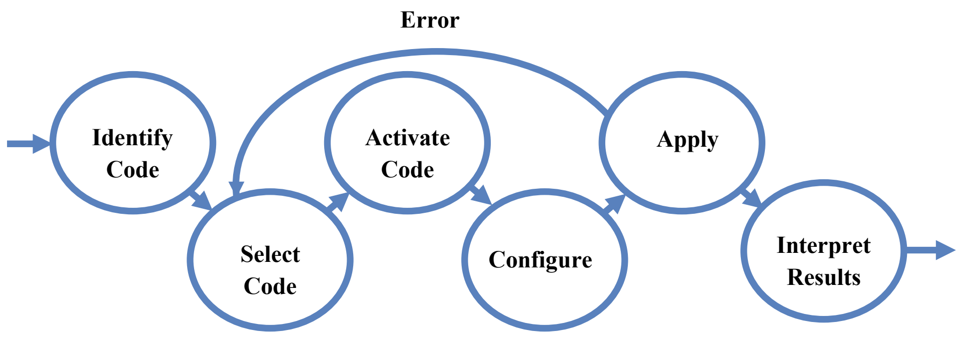


Figure 2.1 Refactoring Process with a Tool

The first step is to identify bad smells in the code. Bad smells are blocks of source code with bad designs in the existing software, which offers opportunities to undertake refactorings (Fowler, 1999; Srivisut & Muenchaisri, 2007). These bad smells can be detected by a variety of symptoms, and subsequently removed to improve the quality of the software. After selecting the code to be refactored, a certain refactoring pattern is activated upon the selected code. Some refactorings need configuration, such as giving a function a new name. Therefore, the third phase is optional. After configuration, refactoring patterns are applied to code. If refactoring is not successful, for example, the new name already exists, then the error is interpreted, and at the same time, the process returns to the "Select Code" phrase.

Refactoring has many patterns. The most popular refactoring patterns include: "Rename" and "Extract Method" (Fowler, 1999). "Rename" is undertaken because the name cannot reveal its purpose, such as renaming "ChangCN" to "ChangeCustomerName". "Extract Method" moves a code fragment into a method whose name indicates the purpose of the method. Murphy-Hill et al. (2009) describe the refactoring patterns in three levels. High-level refactorings are the patterns that change the signatures of classes, methods and fields; medium-level refactorings are the patterns that not only change the signatures of classes, methods and fields, but also significantly change blocks of code; low-level refactorings are the patterns that change only blocks of code. They also state that 40% to 60% refactorings are low and medium level patterns according to two groups of data (Eclipse CVS and Toolsmiths). A variety of refactorings and catalogs of various refactorings are widely available (Fowler, 1999; Thompson & Reinke, 2002; Kerievsky, 2004).

Refactoring can support two automation approaches (Mens & Tourwé, 2004), either semi-automatic or fully-automatic. The semi-automatic approach allows users to decide which patterns need to be applied and which parts of the source code need to be transformed. This approach is more flexible; however, it is time-consuming when the project gets larger and more complex. The fully-automated approach does not require users' participation; the tool will undertake the refactoring automatically. The problem of this approach is that the code after refactoring becomes less understandable and readable. Furthermore, only simple refactoring patterns can be implemented using a fully-automated approach because more complex refactoring patterns require domain knowledge before the refactoring can be applied. For example, the "Rename" pattern requires the programmer to provide the new name that is more easily understandable than the old name. In order for a refactoring tool to be successful, it needs to support many refactoring patterns including the more complex patterns. The semi-automated approach allows the refactoring tool to achieve this objective. It can automatically refactor simple patterns. For more advanced patterns, it interacts with the programmer to obtain needed inputs, then automatically refactor the code to the programmer's requirements. Additionally, the semi-automated approach allows the programmer to still be in control of the refactoring by allowing the programmer to review, edit and approve all refactorings; thus providing outputs that programmers are more likely to trust. This is the reason why many refactoring tools, for example Eclipse's refactoring feature, utilize the semi-automated approach. This is also the reason why the tools presented in this thesis utilize the semi-automated approach.

There are many refactoring techniques. However, the "invariants, pre and post-conditions" refactoring technique (Opdyke, 1992; Roberts, 1999) is the most popular one. The invariants, pre and post-conditions are three different assertions, which are established to indicate the behavior of a program. Invariants are behaviors or properties to be preserved by the transformational process. Pre and post-conditions are required to be satisfied before and after refactoring has been applied.

Unit testing (Coelho et al., 2006) is essential for refactoring; it is adopted to ensure the behavior of a program before and after the refactoring is preserved. A unit is the smallest piece of a program that is testable, such as a method. Refactoring – incorporating unit testing – is designed to assist the programmer with the transformation, while providing safeguards to ensure that defects are not introduced during the transformational process.

2.2 The Flash Execution Model

On the server side, the ActionScript compiler converts an AS3 program into ActionScript bytecode (ABC). This is because compiling from bytecode to machine code is much faster than compiling directly from source code. However, the ABC must be wrapped into a binary container file (.swf file) before it can be executed by the Flash runtime tools such as Flash Authoring Tools, Flash players and browsers (Moock, 2007). The .swf file includes the ABC as well as embedded media assets, such as images, audios and videos.

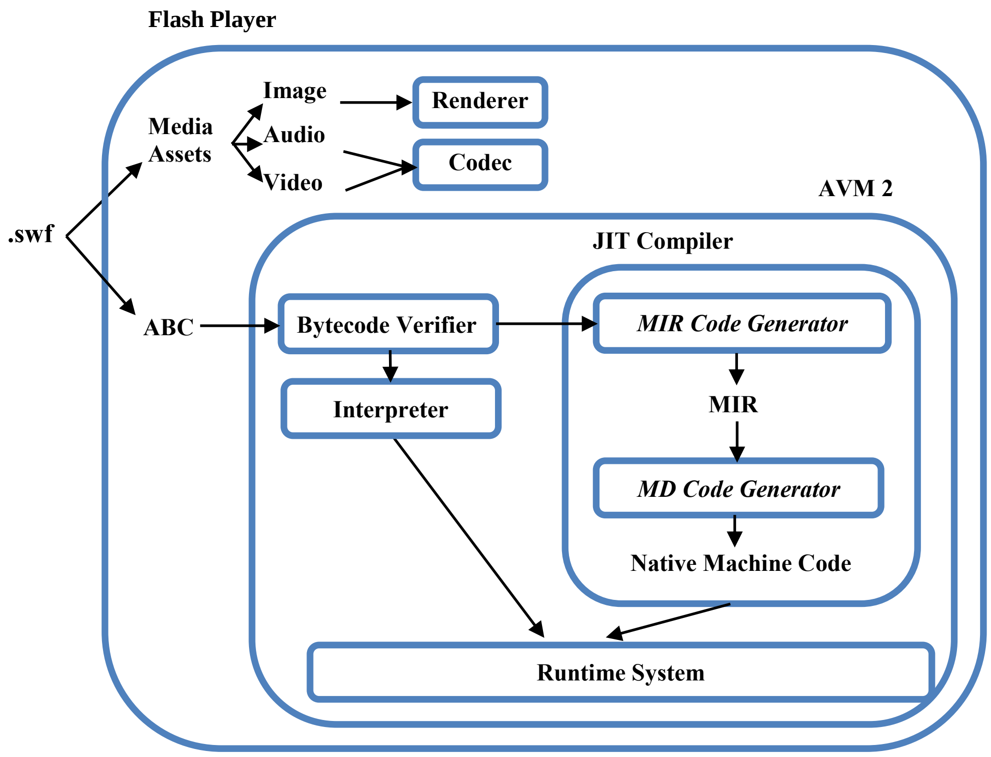


Figure 2.2 Client Side's Flash Execution Model

On the client side (browser), the input to Adobe Flash Player is the .swf file. The client side's Flash execution model is shown in
Figure **2.2** (Grossman, 2006) which consists of several interacting components. The Codec and Renderer process the media assets inside the file. The ActionScript Virtual Machine (AVM) processes the ABC files (the latest version of the AVM is AVM2). In the AVM2, the Bytecode Verifier verifies the ABC code; this includes the verification of the code's structural integrity and type safety (Grossman, 2006). The AVM2 applies a hybrid execution model by either interpreting the ABC directly or invoking of the Just-in-Time (JIT) Compiler. The JIT Compiler translates bytecode into native machine code through two passes.

(1) The Macromedia Intermediate Representation (MIR) Code Generator is used to convert the ABC into a MIR. The MIR is an internal representation between the ABC and the target instruction set. It enables the following optimizations and makes the mapping to the underlying hardware easier (Polanco, 2007).
   - Early Binding: This is used when declaring a variable as a specific object type. Therefore, the verification occurs before the program executes which will save execution time. To take advantage of early binding, typing is very important.
   - Constant Folding: This is the process to simplify constant expressions at compile time. The JIT Compiler firstly searches for unchanged constants, integer values and calculations, and then replaces them with the calculated values.
   - Copy and Constant Propagation: This is the process of replacing constants whose value is known in expressions at compile time to reduce or eliminate redundant code.
   - Common Sub-expression Elimination (CSE): This process searches for identical calculations and makes decisions on whether they can be substituted by a variable to hold the calculated value.

(2) The Machine Code (MD) Generator is used to convert the MIR into platform specific instructions (such as X86, Power PC and ARM). The MD Generator performs the following optimizations.
   - Instruction Selection: This is an algorithm that runs on the MIR and is used to yield a minimized instruction set which implements all the required functionality.
   - Register Allocation: The goal for register allocation is to improve the program execution time by storing as many operands in the register as possible. The JIT Compiler utilizes a linear scan algorithm to achieve register allocation.
   - Dead Code Elimination (DCE): This cleans a program by removing unreachable code without changing the functionality of the program. It reduces the program size and speeds up execution.

Since the AVM2 applies a hybrid execution model, the ABC can be interpreted or compiled. So how does the AVM2 choose between the two? Grossman (2006) (from Adobe Systems) states "initialization functions ($init, $cinit) are interpreted, everything else is JIT". Hence, we can consider that everything is compiled; this assumption is utilized in the reminder of our work.

2.3 Ajax

Ajax is a set of web development technologies to make web applications more interactive and dynamic. Ajax improves the user experience by updating the content within a web application without reloading the entire web page (Paulson, 2005).

In the classic web application model (Figure 2.3), users on the browser (client) side trigger an HTTP request to the server side. The server side deals with the request and returns the updated page to the user. Within the Ajax web application model (Figure 2.4), an Ajax engine runs within the browser to communicate with the server, perform interactions, and display requested information in the browser. If the Ajax engine requires more data, it sends requests asynchronously to the server in the background to retrieve updated data (and potentially additional code) without interfering with the users' interaction with the application (Zakas et al., 2007).

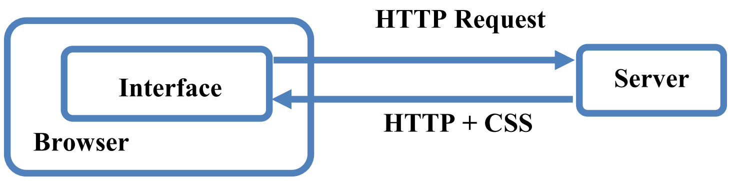


Figure 2.3 The Classic Web Application Model

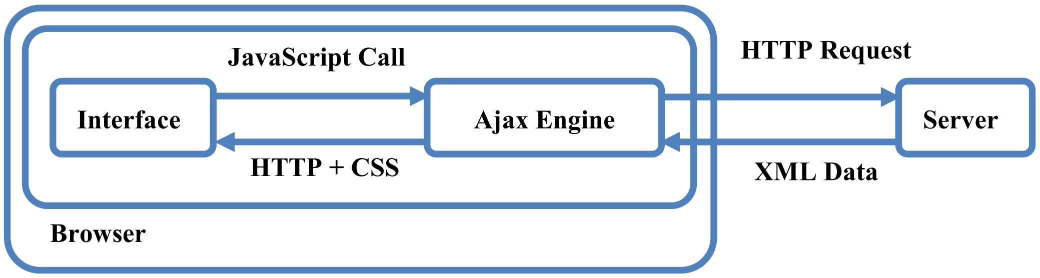


Figure 2.4 The Ajax Web Application Model

Four aspects influence the performance of an Ajax application (Savoia, 2001).

(1) Network transfer time: Network transfer time depends upon the amount of data to be transferred, and the available bandwidth. This latter factor is clearly outside of the programmer's control.

(2) Network latency: Network latency is a combination of:
- The average delay of a zero-byte transfer from the client to the server or vice versa.
- The number of transfers required to complete the request/response cycle.

The number of transfers required is dependent on a number of factors including the version of HTTP(S) that is utilized.

(3) Server processing time: The server processing time reflects the server's capability in handling requests and processing application logic. Data processing time on the server has great influence on the server processing time.

(4) Client processing time: JavaScript processing time on the client is also crucial for Ajax applications, because Ajax technology relies upon a JavaScript-based Ajax engine to interact with the browser. See Figure 2.4.

2.4 XML

Typically, the format for retrieving Ajax data is XML, as the "X" in the name of Ajax indicates. XML, an acronym for Extensible Markup Language, is a subset of the Standard Generalized Markup Language (SGML). XML stores self-describing data in standardized ways, and allows user-defined document markups to be created and formatted (Allen et al., 2008). The primary uses for XML are data interchange and storage in web environments. XML represents data using a hierarchical structure. Figure 2.5 shows a simple example of an XML data structure.

```
<movie>
   <title>A</title>
   <rating>6.5</rating>
</movie>
```

Figure 2.5 An Example of XML Data Structure

The advantages of XML include flexibility and readability. However, XML is not optimal for data interchange between machines because it is overly verbose.

2.5 JSON

An alternative data format to XML is JSON; JSON is short for JavaScript Object Notation. JSON is a lightweight data interchange format based on a subset of the array and object literal notations of JavaScript (Standard ECMA-262). Figure 2.6 shows an example of a JSON data structure, which can be considered equivalent to the previous XML data structure.

```
{
 "movie":{
    "title": "A",
    "rating":"6.5"
  }
}
```

Figure 2.6 An Example of JSON Data Structure

JSON has the advantage of being compact and directly supported by JavaScript. The biggest disadvantage of JSON is that the format is not very readable for humans.

The process for transmitting JSON data between the browser and server is as follows (Webucator, 2009).

1. On the client side: The client creates a JavaScript object and then serializes the JavaScript data structures into JSON text by using JSON stringifier[11] (for JavaScript). After that, the client uses GET or POST methods to trigger an HTTP request, which contains the encoded JSON string.
2. On the server side: After receiving the request, the server deserializes the JSON string into an object by using a JSON parser for the language used by the server. For example, Argo[12] is a JSON parser for the Java programming language. Subsequently, the server manipulates this object for different purposes.

[11] http://www.json.org/json2.js

[12] http://argo.sourceforge.net/

Chapter 3 Related Work

This chapter discusses works related to this dissertation. The chapter is divided into several sections. Sections 3.1 and 3.2 discuss related works that are general to the entire dissertation. Sections 3.3 to 3.4 contain related works that are specific to certain chapters. Chapter 5 does not have any true related works; the topic behind this research is refactoring for software migration and related works on this topic is covered in Section 3.1.

3.1 Related Work on Refactoring

The earliest works on refactoring were primarily for improving the quality of software. For example, Opdyke (1992) defines different refactoring patterns to automatically restructure object-oriented programs. Fowler (1999) provides a comprehensive catalog of refactorings, the principles of refactorings, when and where to implement refactorings for object-oriented programs. Tokuda et al. (2001) discuss how to design object-oriented applications to improve software design. Dudziak and Wloka (2002) provide a method to detect structural weaknesses to improve code structure. Tahvildari and Kontogiannis (2004) propose a reengineering framework to detect potential design flaws by using object-oriented metrics and apply transformations to improve the specific qualities of a software system. Griswold (1991) talks about restructuring programs to improve software maintenance. Refactoring has been implemented in many different programming languages, such as refactoring for C (Garrido and Johnson, 2003), Smalltalk (Roberts, 1999) and Java[13] and UML model (Sunyé et al., 2001).

Refactoring is now being used for different purposes. For example, Weber et al. (2001) propose a catalogue of process model "smells" to identify refactorings and refactoring techniques, so large process repositories can be refactored. Mendonça (2004) presents RefaX, an XML-based refactoring framework to facilitate the development, customization and reuse of refactoring tools. With RefaX, it is possible for programmers to build refactoring tools independently from the source code model, programming language and XML processing technologies. Cinnéide et al. (2011) present an automated refactoring approach to improve the cohesion properties of a program, which in turn has effects on improving the testability of a program.

Some of refactoring approaches are for migration purposes. Lindvall et al. (2003) describe a process to restructure an existing experience management system (EMS) to improve the architecture of the system. Matthews et al. (2001) show how to automatically transform traditional interactive programs into CGI programs. Kjolstad et al. (2009) design an algorithm and present a tool (Immutator) to transform a Java mutable class into an immutable class. Khatchadourian and Muskalla (2010) present a refactoring tool, Convert

[13] http://refactorit.sourceforge.net/

Constants, to transform the legacy Java program to use the new enum construct in Java 5. Roock and Havestein (2002) propose refactoring tags to help developers carry refactorings undertaken upon frameworks to new versions. Kiezun et al. (2007) present an algorithm based on type-constraints to help programmers convert libraries to include type parameters to increase type safety and the expressiveness of libraries. Mancl (2001) introduces several refactorings to allow programmers to reuse and extend existing systems.

Some refactoring approaches for efficiency purposes also exist, such as refactoring for parallelism. Kennedy et al. (1999) present a tool, ParaScope Editor, for parallel Fortran programs. Liao et al. (1999) present an interactive parallelization tool called SUIF Explorer. Wloka et al. (2009) present an Eclipse based refactoring tool (Reentrancer) to make single-threaded Java programs reentrant by replacing global mutable state with thread-local state. Parallel speedups can be enabled through this refactoring. Fuhrer and Saraswat (2009) present several concurrency-related refactorings for the X10 language. Dig et al. (2009b) present a refactoring tool (RELOOPER) to safely refactor a Java array into a ParallelArray which is an array data structure that allows parallel operations. Dig et al. (Dig et al., 2009a; Dig, 2011) present a refactoring tool (CONCURRENCER) to refactor sequential code into concurrent code. Schäfer et al. (2011) present a refactoring tool (Relocker) to assist programmers with the refactoring of synchronized blocks into ReentrantLocks and ReadWriteLocks.

Additionally, Demeyer (2002) refactor C++ programs by replacing conditionals in the programs with polymorphic method calls. The results from their work show that the refactored program is faster than the original program. Méndez et al. (2010) present two categories of Fortran refactorings: refactorings to improve maintainability and refactorings to improve performance. They also developed a refactoring tool (Photran) to implement these refactorings. Beyls and D’Hollander (2009) present a cache profiling tool (SLO) to find the root cause of poor data locality which generates cache misses. The implementation of the refactorings based upon suggestions by SLO could improve the program execution speed.

Different code transformation techniques have been proposed. Bulka and Mayhew (2000) present many optimization techniques for coding and designing of the C++ programming language to improve the code efficiency and performance. Panda et al. (2001) introduce a variety of optimizations for data and memory used in embedded systems from the viewpoint of area, performance, and power dissipation. They discuss architecture-independent optimizations: code-rewriting techniques for access locality and regularity, and code-rewriting techniques to improve data reuse. In addition, they also discuss optimization techniques on different levels of memory architectures, ranging from register files to on-chip memory, data caches, and dynamic memory.

3.2 Related Work on Speeding up Web Applications

There are significant amount of research on speeding up web applications. Some of them are for improving data transfer time and server response time. For example, Myers et al. (2007) present MapJAX, a data-centric framework for accelerating data transfer and callback time of Ajax applications. MapJAX provides an abstraction of logical data structures shared between client and server to replace asynchronous RPCs. Vingraleka et al. (1999) designed Web++, a system for a fast and reliable HTTP service. It improves the response time by dynamically replicating popular web resources among multiple web servers.

Reducing file size is also important for building high performance web sites. JSMin[14] and YUI Compressor[15] are tools for removing comments and white spaces found in JavaScript programs to reduce file size for faster download speed. Page Speed[16] is a tool for improving the performance of web applications by optimizing browser rendering. King (2008) provides ten techniques to maximize web page display speed. Some examples include optimizing JavaScript for execution speed and file size, resizing and optimizing images, and minimizing HTTP requests.

Web caching is another useful technique to accelerate web applications. It can be classified into either server-side or proxy-based caching. In the server-side caching category, Datta et al. (2001) build Dynamic Content Acceleration (DCA), a server-side caching engine that caches dynamic page fragments to reduce dynamic page generation latency on a web site. In the proxy-based caching category, Challenger et al. (2004) present architectures and algorithms for efficiently serving dynamic data at highly accessed web sites. As a result, the system is able to achieve cache hit ratios close to 100% for cached data. The edge server (also refers to client-side proxies, server-side reverse proxies at the edge of the enterprise, or caches within a content distribution network (CDN)) is an architecture to increase the scalability of the back-end and reduce the client response latency. Amiri et al. (2003) describe DBProxy, a self-managing edge-of-network semantic data cache that maintains partial but semantically consistent materialized views of previous query results, to accelerate web applications. Ramaswamy et al. (2007) demonstrate that cache cooperation can significantly improve the performance of edge cache networks. They specifically designed cooperative EC grid - a large-scale edge cache network to deliver highly dynamic web content and support low-cost cooperation among its caches.

Database caching is used for speeding up database-backed web applications. Ghosh and Rau-Chaplin (2006) propose an approach that integrates a HTML fragment cache and a middle-tier database cache to improve performance and scalability. It manages cache storage to reduce response time latency by storing database tables in the middle-tier cache and sharing common fragments among

[14] http://crockford.com/javascript/jsmin.html
[15] http://developer.yahoo.com/yui/compressor/
[16] http://code.google.com/speed/page-speed/

multiple pages stored in a HTML fragment cache. Luo et al. (2008) present a form-based proxy-caching framework for database-backed web servers. They propose two representative caching schemes for web queries: passive query caching which services requests that exactly match the previous requests without any extra processing, and active caching which services requests that can be answered by processing results from previous requests.

There are also some commercial tools for web caching such as the Java Caching System[17] (JCS). This is a distributed caching system for Java applications which can significantly improve Java applications' performance through usage of a cache utility. This cache utility makes it much more convenient to store, access, and delete data in the cache. Squid[18] is a high-performance caching proxy for web users to reduce bandwidth and improve response times by caching and reusing frequently-requested web pages.

3.3 Related Work Specific to Chapter 4

3.3.1 Dynamic Optimization

Dynamic class loading, runtime binding and shared libraries etc. are heavily used in modern software, which make it impossible for static analysis systems to accurately analyze programs. However, even with the help of profilers, what static optimization can achieve is limited. In this situation, shifting optimization to runtime (dynamic optimization) becomes the obvious choice. Dynamic optimization systems typically use caches to buffer optimized hot traces (the frequently executed control flow paths) to benefit from the repeated use of the trace in the code cache. Dynamo (Bala et al., 2000) (for PA-RISC) is a transparent dynamic optimizer to observe run-time behavior without instrumentation. Native instruction stream is the input for Dynamo. Dynamo interprets the instruction stream until a "hot" instruction sequence (trace) is identified for optimization. Some optimizations, such as constant propagation, elimination of redundant branches, and strength reduction are applied before the traces are placed into a trace cache. DynamoRIO[19] is based on Dynamo (for x86 system), the run-time information is not obtained by interpreting instructions, but by the execution of instrumented basic blocks from the basic block cache. The Dynamic Execution Layer Interface (DELI) (Desoli et al., 2002) is a software layer between the application software and the hardware platform. It provides a uniform infrastructure for building client applications that manipulate or observe running programs. The Binary-Level Translation (BLT) layer is the core component, which offers basic code caching, linking service and dynamic code transformations to the client applications. However, it has no mechanism for re-optimizing traces after they are placed into the code cache. Strata (Hiser et al., 2006) is a flexible and adaptable optimization system which focuses on dynamic optimizations. The online optimization plans of Strata are formed at compile time

[17] http://www.ibm.com/developerworks/java/library/j-jcs.html
[18] http://www.squid-cache.org/
[19] http://dynamorio.org/

using both static and dynamic information. According to these plans, optimizations are then applied by the runtime system which is based on a software dynamic translator (SDT). DynamoRIO, DELI and Strata all export APIs for custom optimizations to instrument the traces and basic blocks. Mojo (Chen et al., 2000) (for Windows NT running on IA-32) is a dynamic software optimization system which controls the execution of fragments of code buffered in the Path Cache or the Basic Block Cache to improve the performance of a variety of programs including multi-threaded applications which use exception handling.

Dynamic instrumentation which analyzes or modifies software by inserting trampolines is used for debugging and performance monitoring. Dyninst (Buck & Hollingsworth, 2000) provides an API to insert code into a running program. The running program will keep executing without re-compiling, re-linking or re-starting. Vulcan (Srivastava et al., 2001) is a next generation binary transformation tool that transforms x86, IA-64, or MSIL code into an abstract representation before transforming it back to x86, IA-64, or MSIL code. It provides extensive APIs for both static and dynamic code modification on the abstract representation.

3.3.2 Compiler Optimization

Compiler optimization is a useful approach to reduce the execution time. There are many techniques with regard to compiler optimization, such as loop optimizations, data-flow optimizations, SSA-based optimizations and code generator optimizations. Bacon et al. (1994) provide an overview of important high-level program restructuring techniques for imperative languages, such as C and Fortran. They also discuss where and when these techniques should be applied to high-performance uniprocessors, vector and multiprocessor machines.

However, optimizations that compilers can perform are limited. For example, the GNU C Compiler is an optimized C compiler but only implements the following optimizations: constant folding, common subexpression elimination (CSE), dead code elimination (DCE), function inlining, loop unrolling, scheduling and strength reduction, to produce efficient code (Gough, 2004). The Intel C++ compiler[20] for Windows applies interprocedure optimization (IPO) which is a collection of optimization technologies to replace multiple function calls with actual function code and performs absolute instead of relative addressing. The compiler can also perform profile-guided optimization (PGO) to reorganize code layout to reduce instruction-cache thrashing, code size and branch mispredictions. The JIT compiler in the JVM can only implement optimizations for register allocation and instruction scheduling without any intermediate representation being created. The JIT compiler in AVM2 can do optimizations such as constant folding, copy and constant propagation and common sub-expression. For compiler writers, it is difficult to choose which techniques are to be used to improve efficiency. Therefore, Lee et al. (2006) propose a method for measuring the costs and benefits of compiler optimizations to help compiler writers choose the

[20] http://www.aertia.com/productos.asp?pid=147

optimization methods. Pan and Eigenmann (2006) propose Combined Elimination (CE), a fast and effective compiler optimization orchestration algorithm to tune programs by computing the main effect of the optimization and detecting the interactions between the optimizations. Cavazos et al. (2007) propose a different approach using performance counters to predict good compiler optimization settings. It is achieved by using machine learning techniques to gather performance counter features.

Many compilers make use of adaptive optimization. The HotSpot JVM Runtime Compiler (Hewlett-Packard Company, 2001) provides adaptive optimization. It starts by interpreting all the code and after monitoring the execution of the code, the compiler finds the “hot spot” methods (the 10 % to 20% of the code but occupy 80% to 90% of the execution time), and then compiles these methods and applies optimizations to the native code. It uses method inlining optimization to inline the “hot spot” critical methods to reduce the method invocations and provide more opportunities for code optimization. Therefore, by only compiling “hot spot” code, HotSpot JVM Runtime compiler is able to spend more time on optimizations than a classic JIT. The compiler can also do the following optimizations: class-hierarchy inlining, global value numbering, optimistic constant propagation, optimal instruction selection, graph coloring register allocation and peephole optimization. Jalapeno (Arnold et al., 2000) is a JVM with adaptive optimization. The dynamic optimizing compiler is the key component of the Jalapeno Adaptive Optimization System with a compile-only approach (instead of the interpreter and JIT compiler working together solution). It has three fully operational compilers. The baseline compiler is for translating bytecode programs directly into native code; the optimizing compiler is for compiling computationally intensive methods; and the “quick" compiler is for performing low level optimizations. Jikes RVM[21] (Research Virtual Machine) can perform three levels of optimizations. Level 0: branch opts, constant propagation, DCE, register allocation and instruction scheduling; level 1: pre-existence and speculative inlining, static splitting, CSE, load elimination and flow-insensitive const/copy/type propagation; level 2: loop normalization and unrolling, scalar SSA, dataflow analysis and global CSE.

Some compilations occur before a program executes (static compilation), some occur just before a program is about to run (dynamic compilation or just-in-time compilation). Some are performed at different stages (staged compilation). Staged dynamic compilers delay a part of the compilation until runtime to reduce the cost of run-time code generation while enabling a wide range of optimizations on the code which is generated statically or dynamically. DyC (Grant et al., 1999) is a selective, value-specific dynamic compilation system which speeds up C programs. DyC’s dynamic compilers are staged. Parts of the optimization take place at static compile time without run-time program representation required, and parts of the optimization occur during dynamic compilation. DyC automatically caches the dynamically compiled code; and reuses it to reduce dynamic

[21] http://jikesrvm.org/Jikes+RVM%27s+compilers

compilation overhead. FABIUS (Lee & Leone, 1996) is a compiler that automatically compiles code written in a subset of Metalanguage (ML) into native code. FABIUS is also a staged dynamic compiler which dynamically generates code by using partial evaluation techniques without any intermediate representation created at run time. The dynamic optimization of the Dynamo (Leone & Dybvig, 1997) compiler can be implemented by postponing the remainder of compilation at a certain stage. For example, if a region may benefit from heavyweight (lightweight) dynamic optimizations, it will be partially compiled into a high-level (low-level) intermediate representation. If the code cannot benefit from any code optimizations, then it will be compiled statically.

Some previous works define new compiler architectures to achieve better performance. For example, Briki (Cierniak & Li, 1997) is a Java compiler architecture which focuses on the optimizations that are possible or easier to be implemented on a higher level intermediate representation-JavaIR (Java Intermediate Representation). Briki recovers a high-level structure from the .class file and then applies optimizations to the JavaIR before outputting the optimized code. This way, the computation time is greatly reduced.

3.3.3 Speeding up Embedded System

To improve the performance of compilers for embedded systems, “the basic strategy is to present the algorithm in a way that gives the optimizer excellent visibility of the operations and data”. Analog Devices[22] provides many basic strategies for tuning C programming language for different embedded system processor compilers, such as Blackfin and TigerSHARC DSP. These strategies include: do as much work as possible in the inner loop, use integers for loop control variables and array indices, do not place function calls in loops and avoid conditional code in loops etc. These strategies are like the refactoring patterns in Chapter 4, which cannot be optimized by a compiler, but can make full use of the compiler to improve code efficiency.

Improving cache performance is also essential for embedded systems. Chen et al. (2005) address the problems of developing a cache-less embedded system to reduce the power consumption and improve performance. They encode and wrap the most frequently executed code into pseudo instructions and then use a decompression engine to fetch and extract multiple instructions to reduce memory access time. Kim et al. (2008) propose a data cache for low power and high performance multimedia-oriented embedded systems. They use a small block size direct-mapped cache (DMC) for temporal locality and a large block size fully-associative buffer (FAB) for spatial locality. In addition, they also provide two hardware enhancements: an adaptive multi-block prefetching and an effective block filtering mechanism. Bartolini and Prete (2005) provide a new Cache-Aware Code Allocation Technique (CAT) to improve the cache performance of embedded systems through reducing conflict cache misses. It restructures the

[22] http://www.analog.com/en/index.html

compiled code to optimize locality features of the memory which the cache is able to exploit.

Reducing the code size is another method to improve the performance of embedded systems. Lekatsas and Wolf (1999) present SAMC, a code compression method for reducing the memory requirements of an embedded system. It is only for instruction compression and it is capable of decompressing compressed code during runtime. Seong and Mishra (2006) present a bitmask-based code compression technique which outperforms existing dictionary-based techniques. It has the ability to significantly improve the compression ratio without any decompression overhead. Zmily and Kozyrakis (2006) use a block-aware instruction set (BLISS) which stores basic block descriptors that separate from the actual instructions to achieve three efficiency metrics: smaller code size, better performance and lower energy consumption. It reduces the code size by removing redundant sequences of instructions across basic blocks and by interleaving 16-bit and 32-bit encodings at instruction granularity (Zmily, Killian & Kozyrakis, 2005; Zimily & Kozyrakis, 2005).

Using compiler optimization is another approach to improve the performance of embedded systems. Ghodrat et al. (2007) present a short-circuit code transformation technique which optimizes conditional blocks in high-level programs for embedded systems. Šimunić et al. (2000) present a source code optimization methodology and a profiling tool to optimize software performance and energy in embedded systems. They are used to optimize and tune the implementation of an MPEG Layer III (MP3) audio decoder for the SmartBadge. Three levels of code optimizations are discussed. (1) Algorithmic level: using algorithmic optimization in MP3 decoding. (2) Data level: using fixed-point instead of floating point. (3) Instruction level: using well-known instruction-level techniques (loop merging, unrolling, software pipelining, loop invariant extraction etc.).

3.3.4 Speeding up Distributed System

For distributed systems, load balancing can reduce the job's response time, increase the performance of each host and get the small jobs away from starvation (Jain & Gupta, 2009). Load balancing algorithms have two basic strategies, static (the work load is distributed in the start) and dynamic (the work load is distributed at runtime). In addition, according to the location that initiates load balancing, the algorithms can be classified into source-initiative (transfer jobs to other hosts) and server-initiative algorithm (find jobs from other hosts) (Ali, 2001).

The load balancing algorithm is an important part of a load balancing service. There are three kinds of load balancing services based on different system levels: network-based load balancing, OS-based load balancing and middleware-based load balancing. Compared to the first two, middleware-based load balancing provides the most flexibility. Othman and Schmidt (2001) present a set of middleware-based load balancing service features, such as server-side transparency, extensible load balancing strategy support and run-time control of replica life times, to optimize overall performance, scalability and reliability. Cygnus (Balasubramanian et al., 2004) is an extendible open-source middleware framework which can support adaptive and non-adaptive load balancing strategies. In addition, in order to evaluate load balancing strategies, they also designed LBPerf which is an open-source middleware load balancing benchmarking toolkit.

3.3.5 Summary

In the previous sections, many techniques for speeding up different applications were discussed. Some are applicable to Flash applications, some are not. Reducing the size of Flash files while preserving their functionality can make Flash files load faster. There are many commercial swf compressors, such as Flash Optimizer[23]. However, compressors do not have the ability to accelerate the execution of Flash files.

Web caching can also be used to enhance users' experience of Flash applications. However, it only saves the response time of requested pages when the requested pages are already in the web cache. It cannot speed up the execution of Flash files which is what ART is aimed at achieving.

Dynamic compiler optimization can also be applied to Adobe Flash Player to optimize the execution of the Flash files. Like classic JIT compilation in the JVM, the JIT compilation in AVM2 is also a dynamic compilation process, but it is not adaptive. Adaptive compilation optimizations on the basis of runtime profile information can make AVM2 more efficient. However, the overhead for the dynamic compiler, which spends time on monitoring the execution of a program, selecting which path is a hot trace, etc., is much larger than that of a static compiler. The ActionScript bytecode compiler can also be staged which means a part of the compilation is performed by the static compiler and the other part of the compilation is postponed until runtime to reduce the cost of run-time code generation. ART cannot compete with compiler optimizations which change the way ActionScript bytecode compiles. However, there are many disadvantages of performing optimizations on JIT, as discussed in Chapter 4.

Beyls and D'Hollander's (2009) research is perhaps the closest in concept to the research presented in Chapter 4. They present a cache profiling tool to find the root cause of poor data locality which could generate cache misses by analyzing the runtime reuse paths, and provide the most promising optimizations through three levels: loop, iteration and function. The tool improves efficiency by reducing cache misses; ART improves efficiency by optimizing AS3 language structures used by AS3 programmers.

3.4 Related Work Specific to Chapter 6

Converting XML to JSON is not new. There are many existing tools to convert XML files to JSON files such as the XML to JSON Convertor[24] (xml2json.js).

[23] http://www.show-kit.com/flash-optimizer/

[24] http://www.thomasfrank.se/xml_to_json.html

Tools to convert a XML DOM to a set of JSON objects are also available such as the XML Objectifier (X2J)[25] tool and the XML to JSON Plugin[26] (jQuery.xml2json). However, there are currently no existing discussions or solutions to help web programmers with the conversion of the code statements used to access XML into the JSON equivalent. In other words, current solutions are only half finished; web programmers can use existing tools to convert XML resources into JSON files or objects, but once this conversion is done, they will have to manually rewrite all of the code statements used to process and access the XML resources into their JSON equivalent. Chapter 6 introduced a complete refactoring solution to automatically help programmers with both the conversion of the data structures and the code statements to access these structures.

3.5 Related Work Specific to Chapter 7

Various refactoring proposals for web applications have been discussed. Harold (2008) shows how HTML can be refactored to improve the design of existing web applications. Ricca and Tonella (2001) present a semi-automatic restructuring tool (ReWeb) to implement the analysis on the architecture and evolution of a website. They (Ricca et al., 2002) also present transformation rules on HTML to improve the quality of web applications. Olsina et al. (2007) and Garrido et al. (2011) present a Web Model Refactoring (WMR) approach on the navigation and presentation models. They also demonstrate how to use WebQEM, a quality evaluation method, to test the impact of refactoring. Rossi et al. (2008) present a model-based approach to refactor the web interface of conventional web applications into RIAs. Mesbah and Deursen (2007) propose a migration process to transform multi-paged web applications into single-paged Ajax interfaces. They use a schema-based clustering technique for classifying different web pages and analyzing the candidate's web interface elements for Ajax transformation. Their work focuses on the web interface identification and classification, which is only a starting point of the entire transformation process. Chapter 7 focuses on the source code level transformation from traditional web forms to Ajax-enabled forms. Chu and Dean's research (2008) is perhaps the closest concept to the research presented in Chapter 7. It automatically migrates list based JSP web pages to Ajax web pages using a set of source level transformations. They extract a web service from a JSP page and transform the code of the JSP web page from utilizing the data from a relational database to utilizing XML. Their work is on simple list-based JSP web pages, and they emphasize the transformation of data format and the JavaScript code to access the generated XML file. However, their work cannot transform traditional forms into Ajax-based forms, which is an important part of the interaction between users and the web application. The refactoring process in Chapter 7 is designed to aid web developers transform traditional forms into Ajax-enabled forms including adding validations to the form; hence improving the efficiency and minimizing roundtrip latency for form-based web applications.

[25] http://plugins.jquery.com/project/xmlObjectifier

[26] http://www.fyneworks.com/jquery/xml-to-json/#tab-Overview

Chapter 4 Refactoring ActionScript for Improving Application Execution Time

RIA technologies emphasize a rich and engaging user experience. Graphics, animations and different visual effects are used to create highly dynamic, interactive web pages. However, these pages give rise to serious performance problems. For example, Flash movies and games consist of numerous different graphics (vector and bitmap graphics) which are manipulated to provide a visual experience. After Flash movies or games have been downloaded to the user's machine, these CPU-intensive tasks become the biggest bottleneck and are the principle source of performance problems. If the graphic objects are not well programmed and organized, it will lead to delays or even unresponsiveness. Therefore, RIA client-side technologies require efficient code, such as efficient ActionScript for Adobe Flash, JavaScript for Microsoft Silverlight and Java for JavaFX. How to improve the efficiency of RIAs is a significant challenge, especially given the non-technical background of many programmers in this area and the likelihood of deployment on smartphones.

The user's experience of Flash applications is partially determined by the download and the execution time of Flash files. Download time depends on the size of Flash file and the connection speed to the Internet. The file size can be reduced through the compression of the file. Execution time relies on the processing power of the client machine and the performance of the ActionScript code. Although reducing Flash files' size is helpful, it is not the key point; writing faster and more efficient ActionScript code is the most useful way to improve the user's experience.

The quality of Flash code is highly dependent on the developers; however, Flash programmers often "have backgrounds in music, art, business, philosophy, or just about anything other than programming. This diversity results in awesome creativity and content" (Skinner, 2007), but imposes technical challenges. In addition, the tight schedule of a project tends to result in developers concentrating on "getting the functionality correct" (Skinner, 2007), while ignoring non-functional characteristics such as efficiency.

This chapter presents a "refactoring for efficiency" Flash support system, ART, to help AS3 programmers produce more efficient code by automatically transforming their ActionScript code. This paper is organized as follows. Section 4.1 presents a motivating example; Section 4.2 analyzes possible strategies for improving efficiency in Flash applications. Section 4.3 describes the design of ART. Section 4.4 provides an evaluation of our system.

4.1 Motivating Example

In this section, a motivating example is provided to demonstrate how to improve the efficiency of ActionScript code. Tetris[27] is an open source Flash game developed in AS3. Figure 4.1 shows the interface of the Tetris game.

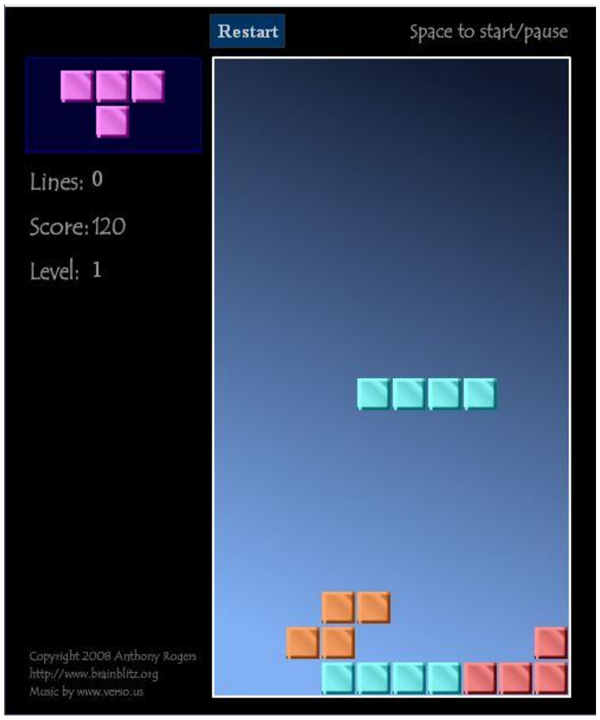


Figure 4.1 The Interface of the Tetris Game

We utilize one function collisionCheckVertical() in the Tetris Flash application to demonstrate our approach. The ActionScript code of the function is as follows.

```
function collisionCheckVertical():void{
  for(ii = 0; ii < body.block.length; ii++){
    if(body.yp + body.block[ii].yp >= cnvMain.height -BLOCKSIZE){
      commitBody();
      return;
    }
  }
  for(jj = 0; jj < aryBlockRow.length; jj++) {
    for(ii = 0; ii < aryBlockRow[jj].length; ii++){
      if(aryBlockRow[jj][ii]) {
        if(body.CollisionCheck(aryBlockRow[jj][ii])){
          commitBody();
          return;
        }
      }
    }
  }
}
```

[27] http://www.4shared.com/file/QVcDsret/TetrisAS30.html

4.2 Three Strategies for Speeding up Flash Applications

By considering the Flash execution model discussed in Section 2.2, we can find three different strategies for improving the efficiency of Flash applications.

4.2.1 Optimizing Bytecode Directly

The first one is to optimize the ABC directly. The bytecode is semantically similar to the source code; however, it is a stack-based, irregular and redundant intermediate representation. The stack-based bytecode causes problems when performing data flow analysis and transforming the code to implement optimizations due to the implicit uses and definitions of stack locations (Bergeron et al., 1999); therefore, many existing optimization techniques are not applicable at this level.

4.2.2 Performing Optimizations on JIT

The second strategy is to perform optimizations when the JIT Compiler generates the binary code. Due to the difficulties of direct stack-based bytecode optimizations, the bytecode is often translated into one (or more) intermediate representation(s), and then into binary code. These intermediate representations are usually stackless (such as register-based) and this enables high-level optimizations and analysis. However, to avoid a considerable startup penalty, the JIT Compiler has to compromise between the time spent on code optimizations and the time spent on program execution. For example, “the Jalapeno VM for Java spends about 93% of its execution time on running application code” (Babic & Rakamaric, 2002). This time requirement makes the implementation of expensive code analysis and optimizations unachievable. Therefore, runtime optimizations for JITs are quite limited. This is why the next generation of compilers employs two JITs: a client side JIT and a server side JIT. For example, the Java HotSpot VM[28] has the Client VM and the Server VM. The client compiler is used to reduce the startup time and memory footprint of applications. Whereas, the server compiler is used to maximize the peak execution speed of long-running server applications which can tolerate higher startup penalties. Similar to the Java HotSpot client VM, the JIT Compiler in the AVM2 performs limited optimizations. The optimizations: early binding, constant folding, copy and constant propagation, and common sub-expression elimination are performed when generating the MIR. Subsequently, when the JIT Compiler’s back-end (the emitter) generates machine code from the MIR, a second limited set of optimizations (instruction selection, register allocation and dead code elimination) are performed. However, expanding these sets of optimizations is problematic as they are always competing with the actual program for resources including CPU cycles.

4.2.3 Refactoring

The third strategy is to perform optimizations offline on the source code through refactoring to speed up Flash applications. Due to the limitations of the previous

[28] http://www.oracle.com/technetwork/java/javase/tech/index.html

two strategies, we have adopted refactoring to make Flash applications faster. Traditional refactoring is mainly for the purpose of readability, extensibility and maintainability. In our situation, code quality is mapped onto efficiency. However, manual refactoring is tedious, error-prone and omission-prone (Dig et al., 2009a); therefore, we have designed ART, a refactoring tool, to improve the efficiency of Flash applications by automatically rewriting AS3 code. ART executes before the ActionScript compiler. Unlike the other two strategies which are online (or during execution) activities, ART is offline which reduces the execution overhead on the client side by delivering already refactored code to the client. It is also more efficient because it only refactors the code once for all the clients that request the same code. Additionally, (1) ART can implement more optimizations than a JIT Compiler; and (2) ART makes no changes to current AVMs or Flash players. Other production approaches also demand refactoring at the source code level. For example, Packager for iPhone which is now a part of Adobe Flash Professional CS5[29] allows Flash developers to deliver applications for the Apple iPhone by reusing the existing AS3 code. The conversion from AS3 to native ARM assembly code is implemented by the Low Level Virtual Machine (LLVM) library. Though the LLVM can perform some code optimizations, these optimizations are limited. ART refactors the source code before the implementation of the LLVM code optimizations to produce compatible yet significantly more efficient code.

4.3 Design of ART

The goal of our research is to build an ActionScript refactoring tool (ART) to make Flash applications faster, so that Flash users' experience is enhanced. However, for ART to be a realistic tool, it needs to consider its impact upon the other characteristics of the system. Negative impacts, if not controlled, may result in ART failing to meet ActionScript programmers' requirements for such a system. Further, there is no point in ART improving the efficiency of a project, while introducing other side-effects (on other characteristics) which jeopardize the success of the project in other directions. Therefore, ART is designed to provide ActionScript programmers with facilities to significantly improve the efficiency of their systems without affecting any other characteristics of the systems.

4.3.1 Characteristics of the System

FURPS[30] is a software quality model developed at Hewlett-Packard. The attributes that impact the software quality include:

1. **F**-Functionality: Feature set, Capabilities, Generality, Security.
2. **U**-Usability: Human factors, Aesthetics, Consistency, Documentation.
3. **R**-Reliability: Frequency/severity of failure, Recoverability, Predictability, **A**ccuracy, Mean time to failure.
4. **P**-Performance: Speed, **Efficiency**, Resource consumption, Throughput, Response time.

[29] http://www.adobe.com/products/flash.html
[30] http://www3.hi.is/pub/honnhug/vika3/furps/tsld002.htm

5. **S**-Supportability: Testability, Extensibility, Adaptability, **Maintainability**, Compatibility, Configurability, Serviceability, Installability, Localizability, Portability.

Positive (beneficial) and negative (adverse) are usually used to indicate the technical interrelationships among these factors. "At the factor level, if factor X positively impacts factor Y, then the presence of factor X will increase the likelihood of achieving the desired quality goal for factor Y. If the indicated relationship is negative, then the presence of factor X will increase the difficulty of achieving the desired quality goal for factor Y" (Lasky & Kevin, 1993; Zulzalil et al., 2008).

For system like ART, the main conflict is between maintainability and efficiency. The relationship between these two attributes is negative which means it is very hard to achieve high efficiency and high maintainability at the same time. For example, modularity positively impacts maintainability while reducing efficiency; however, programming styles for efficient code (optimized or compact code) usually negatively impact maintainability.

4.3.2 Maintainability

Maintainability is a sub-characteristic of the supportability attribute. Maintainability indicates whether a delivered software product has the ability to fix defects, modify and update software components. Industry studies show that over 80% efforts of developing a software product is spent on maintenance (Nelson, 2008). According to ISO 9126[31], among the factors in regard to maintainability, analyzability is essential factor that has the most negative influence on the efficiency of the code. In other words, improving the efficiency of the code will make the code harder to analyze. Analyzability measures the ability to locate failures when bugs occur and the ability to locate modifications when new specifications are added. It is highly influenced by readability, comprehensibility, traceability and simplicity (Spinellis, 2006).

1. Readability

Consistency of coding style is the most important factor that affects the readability of code. The code should be internally consistent as well as externally consistent. Internally consistent requires the similar elements in a program being coded using the same coding style and external consistency requires the program being coded following one of the existing coding styles. There are other factors that influence the code readability, such as the formatting and naming conventions.

[31] http://www.iso.org/

2. Comprehensibility

The process of reading and comprehending a piece of code is a cognitive process. In cognitive psychology, memory is classified into three storage systems (Goldstein, 2007).

(1) Sensory memory: It retains the information provided by visual or auditory, but it only lasts for a few seconds.
(2) Short-term memory: It retains a small amount of information which lasts 3 to 20 seconds.
(3) Long-term memory: It is a relatively permanent storage.

The short-term memory is related to the way programmers understand the program. Miller (1956) indicates that the storage capacity limitation for the short-term memory is seven pieces of independent information plus or minus two. A more recent research shows a lower limitation, about four to five items. The result is heavily dependent on the people being tested and the material being used during the testing (Cowan, 2000). Regardless of the exact number, researchers agree that the limitation is "extremely small". To expand the ability of learning and remembering, chunking is used which allows people to "sequence" and organize information into meaningful groups (chunks). Once a chunk matches an abstraction stored in long-term memory, the information will be removed from short-term memory and replaced by the abstraction. For example, the telephone number is usually chunked into three groups.

Because of the limitation of the short-term memory, a shorter code piece is usually more readable than its longer alternative. Thus, if the number of operands and operators in an expression, or the number of statements in a function or a method exceed the short-term memory storage limitation, a programmer needs to chunk the program which makes the cognitive process more complex. In addition, the storage limitation of short-term memory and the ability of chunking vary from person to person. Thus, it is a good practice to make the length of expressions, functions or methods shorter to improve the understanding and hence, the analyzability of a code piece. There are other factors that affect the code comprehensibility such as the comments for the code blocks and data declaration.

3. Traceability

Tracing is where programmers scan the program back and forth to find the location that they want to modify. Traceability is the degree to which programmers locate dependencies between elements. The locality of dependencies and ambiguity has great impact on code traceability. Couplings and polymorphism are the possible causes of the code ambiguity. Traceability is usually regarded as an element of reviewability. Software's reviewability is a related concept - how easy it is for other programmers to examine the code to ensure that all specifications have been implemented. This is another important concern during the maintenance process. The ease of understanding of the code has a direct impact on the ability to carry out an effective code review (Faris, 2006).

4. Simplicity

When programmers write code, they can use different ways to create different algorithms and functionality; however, clear and concise code is always the best choice (Faris, 2006). This is commonly referred as KISS (Keep it simple, stupid) (Derezińska et al., 2010). The KISS principle is especially popular in Agile circles. Simple code helps both the reviewers and maintainers understand the functionality of the code; hence, it is easier to fix defects and expand the code base. In addition, simple software design, implementation and coding make the software more reliable and bug-free. Therefore, "the ease of maintenance of any piece of software is directly proportional to the simplicity of the individual pieces" (Kanat-Alexander, 2008).

4.3.2.1 Efficiency

Efficiency is a sub-characteristic of the performance attribute. It is about the usage of system resources (memory, network, disk space and etc.) when providing the required functionality. Currently, our refactoring patterns are about increasing the performance of an application.

To summarize, traditional refactoring restructure the code to improve readability, expandability and maintainability. Our refactoring is to improve the efficiency of the code. Our refactoring patterns must not affect the maintainability (readability, comprehensibility, reviewability and simplicity) of the software products.

4.3.3 ART's Execution Model

There are two options for ART's execution model; the first option is to integrate ART and the ActionScript compiler, as shown in Figure 4.2. This model takes AS3 code as the input and outputs the ABC. It is commonly used for complex code transformations because the code after complex transformations is usually unreadable due to the inconsistence of the coding style. However, the enhanced AS3 code after refactoring is not transparent to the programmers in this model. In addition, ART runs every time when the code is compiled into the ABC; thus, if the size of the code is large, it is time-consuming to compile a single file. In this scenario, the usability of the tool is significantly affected.

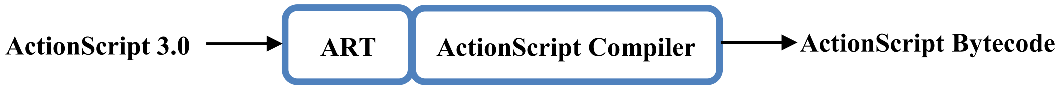


Figure 4.2 One Option for ART's Execution Model

The second option is to separate ART and the ActionScript compiler (as shown in Figure 4.3); this is our choice. In this model, the enhanced AS3 code after refactoring is transparent to the programmers. This is because our code transformations are designed to follow the KISS principle (Derezińska & Sarba,

2010) as commonly practiced in the Agile circles. Additionally, the transformations do not affect the readability and comprehensibility of the code. Now, ART only runs after significant alternations are made to the code base rather than every time it is recompiled. Many refactoring tools also make use of this model, such as the refactoring tool in Eclipse[32].

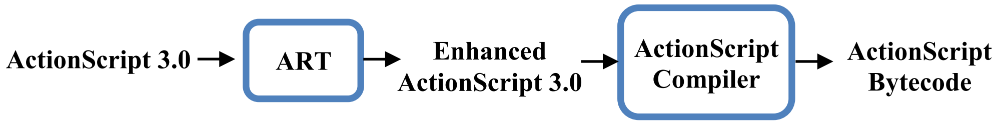


Figure 4.3 ART's Execution Model

4.3.4 Overview of Our Refactoring Cycle

The refactoring process contains two main steps: bad smells (inefficient coding patterns) (Fowler, 1999; Srivisut & Muenchaisri, 2007) detection and code rewriting. Each of the steps can be accomplished by using one of three approaches: fully-automatic, semi-automatic or manual. ART adopts a fully-automatic approach to detect bad smells in AS3 and semi-automatic approach to interact with users (get inputs from users and ask users' permissions to change the code) to implement rewriting. Using the semi-automatic approach to translate bad smells into more efficient and semantically identical code equivalents is required because:

1. A fully-automatic approach makes the refactored code less readable which causes problems to code review and maintenance.
2. Refactoring tools are not smart enough to perform the refactoring in-line with users' wishes.
3. Many refactoring patterns are too complex to allow them to be fully-automatic.

ART is implemented using Another Tool for Language Recognition (ANTLR)[33], a recursive parser generator for building translators, compilers and interpreters. We used ANTLR to build our AS3 parser, because (Kaplan, 1999): (1) it is open source. (2) It supports selective lookahead LL(*) parsing and predicates to resolve ambiguities. (3) It is easier to use than other similar tools and the parser code generated is relative easy to understand, which helps debugging. (4) It generates tree parser without assistance from other tools. (5) It has a good error reporting.

Through ANTLR, an AS3 Lexer, an AS3 Parser and an AS3 Tree Walker are generated using:

AS3 grammar: It follows the specification of ECMAScript, which contains the grammar for tokens, lexer and parser.

[32] http://www.eclipse.org/
[33] http://www.antlr.org/

AS3 tree grammar: Actions (translation rules) are embedded into the tree grammar to implement translation.

Our refactoring cycle consists of six phases, as shown in Figure 4.4 (Parr, 2007).

1. The Lexer scans a character stream and generates a token stream with vocabulary symbols.
2. The Parser constructs an intermediate hierarchical data structure (abstract syntax tree (AST)) from the token stream.
3. The Tree Walker walks the AST.
4. If the Tree Parser finds a bad smell, it asks for inputs from users, and then constructs the required solution using the users' definitions.
5. The Tree Parser rewrites the code by substituting the bad smell for its solution.
6. Go back to the phase 4 to continue searching for the other bad smells until the user has considered them all.

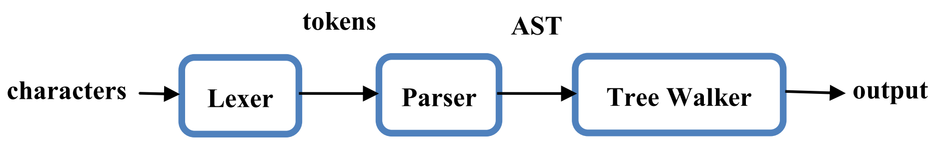


Figure 4.4 Refactoring Cycle

4.3.5 Bad Smells and Refactoring Solutions in ActionScript 3.0

A bad smell and refactoring solution form a refactoring pattern. To begin, we need to know whether our refactoring patterns will be interpreted or compiled. As mentioned previously, only initialization functions ($init, $cinit) are interpreted, everything else is compiled by the JIT Compiler. Therefore, if a refactoring pattern is inside a class constructor, then it will be interpreted; if not, the JIT Compiler will be used to compile the code. Hence, we need to know what kind of code transformations the JIT Compiler performs (as stated in Section 2.2) to make sure our refactorings perform different optimizations.

Our refactoring uses the invariants, pre and post-conditions refactoring technique (Opdyke, 1992; Roberts, 1999). ART is able to check the pre and post-conditions: the syntax of the AS3 code, before and after refactoring, is functionally correct.

Currently, we have identified 43 refactoring patterns and we define our patterns following a pattern template, as Figure 4.5 shows.

Pattern name: The name of the pattern.
Problem: The problem statement including the low efficiency reasons for the bad smells.
Solution: The corresponding refactoring solution(s) for the bad smells.
Input: The user's inputs or permissions to change the bad smells. (The input is displayed in bold in the example.)
Recommend running environments: The recommended browser and Flash Player.
Example: An example of the bad smells and the corresponding refactoring solutions.
Grammar: ActionScript grammar before and after refactoring.

Figure 4.5 The Template of the Refactoring Patterns

Our patterns fall into several categories, the detailed discussion of the refactoring patterns can be found in Appendix A.

1. **Variables Refactoring Patterns:** Variables are used to store values in a program. The var statement is required to declare a variable, for example:

   ```
   var variableName:datatype;
   ```

   Variable declaration style has great influence on the speed of code. Therefore, code transformation is necessary.
2. **Objects Refactoring Patterns:** Objects are frequently used in AS3. This category has three sub-categories: Math and operators, arrays, and others. Math operations are quite useful to draw graphics; however, calling Math objects is inefficient because Math objects are top-level objects in AS3; thus replacements for Math objects are required. In addition, in AS3, when using numbers of type int or uint, bitwise operators are faster than traditional math operators because the bitwise operators allow low-level access to the memory which results in faster execution. Thus, the traditional math operators should be replaced by the bitwise operators. These replacements can be done by strength reduction which is a compiler optimization technique used to replace costly operations by equivalent, but less expensive, operations. However, the JIT compiler in AVM2 cannot do this kind of optimization. Arrays are also ubiquitous in AS3; they are commonly used to store graphic objects. Array manipulations are usually slow; thus it is also a vital area to do the refactoring.
3. **Conditions Refactoring Patterns:** The conditional statements execute different blocks of code depending on whether the condition evaluates to be true or false. The if statement is the most popular conditional statement in AS3. The syntax for an if statement in AS3 is:

   ```
   if(textExpression){
     codeBlock
   }
   ```

   Small modifications to the structure of an if statement (textExpression and codeBlock) can provide faster execution of code. Thus, the conditional statements should be refactored for the efficiency purpose.

4. **Loops Refactoring Patterns:** Loops are widely used in AS3, which heavily affect code efficiency. If a loop executes thousands of times, small changes to the loop structure can significantly improve the code performance. This category has three sub-categories: Pre-calculation refactoring patterns, for loop refactoring patterns and others. The loop-invariant computations inside the loop, which are independent of the iteration of loop, are the redundant computations that affect the efficiency of a program. The simple refactoring here is to: (a) define the variables; and (b) execute the loop-invariant computations outside the loop. This is a well-known optimization for compilers, called loop-invariant code motion; however, the JIT compiler in AVM2 cannot do this kind of optimization. The for loop is the most commonly used loop structure following the format:

   ```
   for (counter; condition; action){
       statements;
   }
   ```

   The counter and the condition of an for loop can be refactored to improve the efficiency of the for loop.
5. **Packages, Classes and Functions Refactoring Patterns:** AS3 are composed of classes, which are the blueprints for the objects of a program. To use a class that is inside a package, the import statement should be used. The format is as follows:

   ```
   import packageName.className;
   ```

   Variables, constants, and methods can be defined within the definitions of a class. A function is a block of code that performs specific task. AS3 has two kinds of functions: methods and function closures. If a function is defined within the class definition or is attached to an instance of an object, the function is called a method. A function is named a function closure if it is defined in some other ways. This category discusses refactoring patterns on the importation of packages and the declaration of classes and functions.
6. **Graphic Display Refactoring Patterns:** Flash movies and games consist of numerous different graphics which are manipulated to provide a visual experience. All the graphics are created, displayed and manipulated by the display list. The display list is a tree structure and the branches and leaves of the display list are different display objects which derive from the DisplayObject class. For example, flash.geom.Point, flash.display.Shape and flash.display.Sprite are different display classes deriving from the DisplayObject class in AS3. In this category, some refactoring on display objects, which has the ability to speed up the Flash applications, will be discussed.
7. **Event/Event Handling Refactoring Patterns:** In AS3, an event is an occurrence that triggers a response. Each event is represented by an event object. To respond to specific events, the event listener function is used.

The event listener function is called when an event is triggered. The syntax of registering an event listener is as follows:

```
addEventListener(TypeOfEvent.NameOfEvent,
             NameOfEventHandlerFunction);
```

Changing the type of the event from TimerEvent.TIMER to Enter.Enter_FRAME and adding the weak reference parameter to addEventListener() have great impact on the efficiency of AS3 programs.

4.4 Evaluation

4.4.1 Methodology

After identifying the bad smells in AS3, we test the performance of our refactoring patterns in different configurations to guarantee the patterns' performance. When testing Flash applications, three aspects must be considered.

1. The Flash authoring tool used.
2. The version of Adobe Flash Player as set in the Publish Settings (PS).
3. The available runtime environments.

Tests are executed in the following environment: Intel(R) Core (TM) 2 Quad CPU Q6600 @2.4GHz, with 4 GB of RAM running Microsoft Windows XP Professional, Service Pack 3. We use getTimer() function in ActionScript flash.utils package to measure execution time. Usually, a timer will return two kinds of time, CPU time or wall time. CPU time contains two parts of time: user time, the time spends on running the actual machine instructions of a program, and kernel time, the time spends in the kernel. Wall time includes CPU time, I/O time and communication delay time. The getTimer() function returns wall time; however, the execution of refactoring patterns does not contain I/O operation and network communication time. Thus, we actually measure the CPU time of the function's execution.

4.4.2 Testing the Performance of Refactoring Patterns

To illustrate the performance of ART, we test the performance of our refactoring patterns. We create the test code before and after refactoring which only contains the bad smell and the solution of an refactoring pattern to minimize the impact of the extended code blocks. We place the code before and after refactoring in different loop structures which interate 1,000,000 times and we run the test program 50 times to obtain an average value. For the patterns with array structure, we use an array with 80 elements. Table 4.1 shows the execution time (in milliseconds) of the original (slow) and the refactored (fast) code for these trials. We use Adobe Flash CS3 Professional (CS3) and Adobe Flash CS4 Professional (CS4) as Flash authoring tools; Adobe Flash Player (9 and 10) as the option for Publish Settings (PS); and Flash authoring tools (CS3 and CS4), Adobe Flash Player (9 and 10) and popular browsers (Internet Explorer 8.0 and Firefox 3.6) as runtime environments. A variation of the relative mean difference (RMD) of the execution time before and after refactoring for each pattern is calculated (Table 4.1). The value (RMD) is calculated as:

$$\frac{\textit{(Execution Time before Refactoring - Execution Time after Refactoring)}}{\textit{(Execution Time before Refactoring + Execution Time after Refactoring) /2}} \qquad (1)$$

If the RMD is positive, then the refactorings have increased the efficiency of code. However, if the RMD is negative, then the refactorings actually cause a decrease in performance.

There are some patterns that cannot be measured (Pattern 23, 36, 40 and 43) by using the getTimer() function. These refactoring patterns offer improvements indirectly through other optimizations (such as better memory allocation and better package utilization) that would require specialized evaluation techniques for each refactoring. Hence, they are not included in Table 4.1 which measures execution time.

As can be seen from Table 4.1:

1. Our refactoring patterns have the ability to improve the efficiency of ActionScript code. Take Math.abs(n) for example, there are multiple versions of refactored code: n<0?(n*(-1)):n, n<0?(-n):n and if(n<0)n=-n. However before testing, we had no evidence about which version is superior. According to the results: (1) all three versions of refactored code are about 20 times faster than the original code. (2) "if statement" version performs better than the other two. (3) The RMD of the execution time between the original code and "if statement" version (the version of refactored code with the best performance) is around 1.75.
2. Different configurations have an influence on the effectiveness of each refactoring pattern. Take "Avoid array.length in for statements" for example, no matter which Flash authoring tool is used, the original code running in Adobe Flash Player 9 takes about 10 times longer than in Adobe Flash Player 10 for both Internet Explorer 8.0 and Firefox 3.6, however, the refactored code running in Adobe Flash Player 9 takes only about 2 times more than in Adobe Flash Player 10 for both Internet Explorer 8.0 and Firefox 3.6. Therefore, this pattern works better in Adobe Flash Player 9 (around 1.7 RMD) than Adobe Flash Player 10 (around 0.9 RMD).

Table 4.1 Testing Results for Different Patterns in Different Configurations

		Timing for Different IDE			**Timing for Different Adobe Flash Player**			**Timing for Different Browser**					
		CS3	**CS4**		**Adobe Flash Player 9**		**Adobe Flash Player 10**	**Internet Explorer 8.0**			**Firefox 3.6**		
		PS for Adobe Flash Player 9	**PS for Adobe Flash Player 9**	**PS for Adobe Flash Player 10**	**CS3 PS for Adobe Flash Player 9**	**CS4 PS for Adobe Flash Player 9**	**CS4 PS for Adobe Flash Player 10**	**CS3 Adobe Flash Player 9**	**CS4 Adobe Flash Player 9**	**CS4 Adobe Flash Player 10**	**CS3 Adobe Flash Player 9**	**CS4 Adobe Flash Player 9**	**CS 4 Adob e Flash Player 10**
	Slow	46.10	55.00	55.16	20.38	20.46	22.26	49.30	49.00	24.70	49.44	51.16	25.86
1	Fast	1.24	1.00	1.02	0.68	0.68	0.64	1.50	1.50	0.62	1.08	1.10	0.72
	RMD	***1.90***	***1.93***	***1.93***	***1.87***	***1.87***	***1.89***	***1.88***	***1.88***	***1.90***	***1.91***	***1.92***	***1.89***
	Slow	3.80	5.84	5.40	1.74	1.72	2.42	5.02	5.34	1.88	5.90	5.86	2.34
2	Fast	3.00	4.62	4.62	1.74	1.72	1.88	4.20	4.18	1.56	4.22	4.18	1.96
	RMD	***0.24***	***0.23***	***0.16***	***0.00***	***0.00***	***0.25***	***0.18***	***0.24***	***0.19***	***0.33***	***0.33***	***0.18***
	Slow	110.62	111.56	110.44	63.30	63.52	59.22	108.8	115.66	55.32	111.30	110.00	59.76
	Fast1	5.60	6.68	16.22	5.42	5.44	5.16	6.40	16.20	5.32	14.94	7.14	5.24
	RMD	***1.81***	***1.77***	***1.49***	***1.68***	***1.68***	***1.68***	***1.78***	***1.51***	***1.65***	***1.53***	***1.76***	***1.68***
3	Fast2	5.56	6.26	16.20	5.44	5.42	5.46	6.28	16.20	5.30	14.94	6.72	5.38
	RMD	***1.81***	***1.79***	***1.49***	***1.68***	***1.69***	***1.66***	***1.78***	***1.51***	***1.65***	***1.53***	***1.77***	***1.67***
	Fast3	5.40	5.44	5.44	5.00	5.02	5.00	5.00	5.42	4.70	5.22	5.86	4.62
	RMD	***1.81***	***1.81***	***1.81***	***1.71***	***1.71***	***1.69***	***1.82***	***1.82***	***1.69***	***1.82***	***1.80***	***1.71***
	Slow	116.78	115.12	125.02	73.62	74.08	67.10	119.20	114.50	65.00	113.56	116.46	69.62
4	Fast	17.54	10.26	15.14	8.92	8.92	8.44	18.78	23.40	8.44	13.76	11.20	8.44
	RMD	***1.48***	***1.67***	***1.57***	***1.57***	***1.57***	***1.55***	***1.46***	***1.32***	***1.54***	***1.57***	***1.65***	***1.57***
5	Slow	207.88	203.74	215.05	106.78	106.34	98.28	230.40	231.80	96.58	219.66	223.60	97.66

	Fast1	142.78	141.40	134.28	85.88	85.26	80.70	152.62	153.02	80.92	147.16	145.94	80.78
	RMD	***0.37***	***0.36***	***0.46***	***0.22***	***0.22***	***0.20***	***0.41***	***0.41***	***0.18***	***0.40***	***0.42***	***0.19***
	Fast2	188.94	198.66	198.16	84.62	85.96	79.62	193.08	192.86	80.32	191.12	191.74	79.92
	RMD	***0.10***	***0.03***	***0.08***	***0.23***	***0.21***	***0.21***	***0.18***	***0.18***	***0.18***	***0.14***	***0.15***	***0.20***
	Slow	121.4	127.38	121.84	75.84	76.98	70.24	122.28	119.86	69.08	119.00	127.88	70.86
6	Fast	14.62	11.70	11.68	11.70	11.70	11.74	14.62	14.62	11.56	11.72	11.70	11.72
	RMD	***1.57***	***1.66***	***1.65***	***1.47***	***1.47***	***1.43***	***1.57***	***1.57***	***1.43***	***1.64***	***1.66***	***1.43***
	Slow	119.88	120.50	119.34	72.62	72.92	67.98	115.48	113.42	64.38	112.14	115.10	67.66
7	Fast	10.16	10.10	20.76	8.60	8.92	8.44	18.80	23.36	8.76	13.78	11.30	8.50
	RMD	***1.69***	***1.69***	***1.41***	***1.58***	***1.56***	***1.56***	***1.44***	***1.32***	***1.52***	***1.56***	***1.64***	***1.55***
	Slow	156.06	140.76	133.62	110.22	110.76	101.00	175.84	175.54	87.82	181.32	178.64	90.62
8	Fast	3.34	4.60	4.60	3.34	3.34	3.36	4.58	4.60	3.44	4.60	4.62	3.34
	RMD	***1.92***	***1.87***	***1.87***	***1.88***	***1.88***	***1.87***	***1.90***	***1.90***	***1.85***	***1.90***	***1.90***	***1.86***
	Slow	105.66	105.62	100.98	62.30	61.84	57.72	111.58	109.00	50.94	114.56	111.36	53.04
9	Fast	5.42	5.86	12.54	10.42	10.28	5.48	5.98	12.56	5.62	6.02	12.54	5.40
	RMD	***1.80***	***1.79***	***1.56***	***1.43***	***1.43***	***1.65***	***1.80***	***1.59***	***1.60***	***1.80***	***1.60***	***1.63***
	Slow	106.70	103.66	104.28	62.72	62.36	58.06	111.12	108.60	50.94	113.38	112.34	53.90
10	Fast	5.46	5.86	12.64	10.40	10.42	5.38	6.00	12.54	5.32	6.04	12.54	5.40
	RMD	***1.81***	***1.79***	***1.57***	***1.43***	***1.43***	***1.66***	***1.80***	***1.59***	***1.62***	***1.80***	***1.60***	***1.64***
	Slow	11.68	7.82	7.90	6.00	6.02	6.02	20.34	20.32	5.64	7.92	7.94	6.16
11	Fast	5.16	4.20	4.16	5.00	5.02	5.00	4.60	4.58	5.00	4.58	4.60	5.00
	RMD	***0.77***	***0.60***	***0.62***	***0.18***	***0.18***	***0.19***	***1.26***	***1.26***	***0.12***	***0.53***	***0.53***	***0.21***
	Slow	35.64	36.38	43.64	38.44	38.42	45.40	37.50	42.62	45.00	46.36	36.44	45.4
12	Fast	2.42	4.16	4.60	3.68	3.62	2.50	4.62	4.62	2.5	4.58	4.62	3.04
	RMD	***1.75***	***1.59***	***1.62***	***1.65***	***1.66***	***1.79***	***1.56***	***1.61***	***1.79***	***1.64***	***1.55***	***1.75***

13	Slow	71.48	59.06	59.90	71.26	71.36	79.18	72.62	73.50	35.12	61.98	59.44	35.14
	Fast	13.84	9.56	10.60	12.36	12.36	8.52	22.88	14.70	7.12	10.62	9.28	7.08
	RMD	***1.35***	***1.44***	***1.40***	***1.41***	***1.41***	***1.61***	***1.04***	***1.33***	***1.33***	***1.41***	***1.46***	***1.33***
14	Slow	219.54	206.4	207.66	166.86	167.32	148.44	240.68	235.76	162.20	223.52	225.64	150.24
	Fast	64.98	61.68	62.86	60.84	61.30	57.66	66.94	70.66	58.12	69.08	68.70	58.52
	RMD	***1.09***	***1.08***	***1.07***	***0.93***	***0.93***	***0.88***	***1.13***	***1.08***	***0.94***	***1.06***	***1.07***	***0.88***
15	Slow	1410.80	1639.80	1507.00	1141.80	1132.40	1151.20	1495.20	1497.40	633.80	1491.20	1496.80	663.80
	Fast	476.20	513.80	517.60	472.20	472.80	499.20	503.20	503.20	283.00	512.60	511.20	280.20
	RMD	***0.99***	***1.05***	***0.98***	***0.83***	***0.82***	***0.79***	***0.99***	***0.99***	***0.77***	***0.98***	***0.98***	***0.81***
16	Slow			4744.00			3579.80			2264.40			2251.40
	Fast			156.60			165.80			84.20			85.00
	RMD			***1.87***			***1.82***			***1.86***			***1.85***
17	Slow	54.46	52.04	52.32	52.40	52.64	51.26	55.60	57.42	50.64	54.96	56.18	50.46
	Fast	46.68	46.36	46.44	44.98	44.46	43.58	47.44	48.50	45.94	47.88	48.52	44.30
	RMD	***0.15***	***0.12***	***0.12***	***0.15***	***0.17***	***0.16***	***0.16***	***0.17***	***0.10***	***0.14***	***0.15***	***0.13***
18	Slow	88.56	70.08	70.80	81.24	81.50	67.26	92.30	92.28	67.82	85.86	85.28	67.88
	Fast	66.64	47.80	46.82	67.10	67.04	45.56	58.18	57.88	46.56	57.36	57.30	45.70
	RMD	***0.28***	***0.38***	***0.41***	***0.19***	***0.19***	***0.38***	***0.45***	***0.46***	***0.37***	***0.40***	***0.39***	***0.39***
19	Slow	146.76	73.94	74.18	138.72	138.72	73.28	157.14	157.10	68.44	150.66	149.86	68.90
	Fast	96.70	151.36	173.26	27.00	27.00	26.32	97.26	95.90	26.56	99.20	98.72	26.18
	RMD	***0.41***	***-0.69***	***-0.80***	***1.35***	***1.35***	***0.94***	***0.47***	***0.48***	***0.88***	***0.41***	***0.41***	***0.90***
20	Slow	68.82	60.96	59.24	64.64	64.74	56.48	74.92	70.14	55.64	69.54	68.86	56.48
	Fast	46.46	42.22	42.58	44.38	44.32	39.78	48.20	48.20	40.30	46.64	46.52	39.78
	RMD	***0.39***	***0.36***	***0.33***	***0.37***	***0.37***	***0.35***	***0.43***	***0.37***	***0.32***	***0.39***	***0.39***	***0.35***
21	Slow	771.00	720.00	735.00	774.00	767.00	711.00	816.00	826.00	672.00	806.00	797.00	684.00
	Fast	513.00	471.00	511.00	362.00	366.00	328.00	571.00	556.00	313.00	550.00	530.00	398.00
	RMD	***0.40***	***0.42***	***0.36***	***0.73***	***0.71***	***0.74***	***0.35***	***0.39***	***0.73***	***0.38***	***0.40***	***0.53***
22	Slow	13690.00	13344.00	13374.00	13582.00	13634.00	12596.00	13856.00	13914.00	11314.00	14595.00	14424.00	11300.00
	Fast	2244.00	2190.00	2156.00	2034.00	2012.00	1856.00	2364.00	2448.00	844.00	2316.00	2308.00	814.00
	RMD	***1.44***	***1.44***	***1.44***	***1.48***	***1.49***	***1.49***	***1.42***	***1.40***	***1.72***	***1.45***	***1.45***	***1.73***

24	Slow	6.72	10.04	9.20	6.30	6.28	5.88	10.08	8.82	5.44	9.66	8.38	5.86
	Fast	3.76	5.02	5.48	3.38	3.42	3.88	6.28	5.44	3.50	5.90	5.00	3.42
	RMD	***0.56***	***0.67***	***0.51***	***0.60***	***0.59***	***0.41***	***0.46***	***0.47***	***0.43***	***0.48***	***0.51***	***0.53***
25	Slow	3.86	7.10	6.70	4.20	3.60	6.70	7.08	3.76	6.72	7.10	4.30	6.70
	Fast	2.92	5.42	5.44	2.96	2.92	5.44	5.44	3.12	5.48	5.44	2.90	5.44
	RMD	***0.28***	***0.27***	***0.21***	***0.35***	***0.21***	***0.21***	***0.26***	***0.19***	***0.20***	***0.26***	***0.39***	***0.21***
26	Slow	5.86	5.94	5.98	5.84	5.84	6.72	5.88	5.86	6.26	5.98	6.00	6.56
	Fast	2.94	4.58	4.60	3.06	3.06	5.38	4.16	4.18	5.30	4.28	4.26	5.46
	RMD	***0.66***	***0.26***	***0.26***	***0.62***	***0.62***	***0.22***	***0.34***	***0.33***	***0.17***	***0.33***	***0.34***	***0.18***
27	Slow	55.80	51.96	53.28	51.34	51.54	51.64	53.00	51.26	49.38	56.26	54.06	51.05
	Fast	2.50	4.20	4.18	2.56	2.52	2.50	4.58	4.62	2.50	4.62	4.68	2.50
	RMD	***1.83***	***1.70***	***1.71***	***1.81***	***1.81***	***1.82***	***1.68***	***1.67***	***1.81***	***1.70***	***1.68***	***1.81***
28	Slow	43.52	48.16	50.32	27.64	28.06	28.12	51.22	45.82	27.5	49.34	46.64	89.56
	Fast	4.38	4.26	1.68	2.66	2.66	2.66	3.46	1.72	2.50	1.82	3.92	2.64
	RMD	***1.63***	***1.67***	***1.87***	***1.65***	***1.65***	***1.65***	***1.75***	***1.86***	***1.67***	***1.86***	***1.69***	***1.89***
29	Slow	72.78	64.08	70.06	69.92	69.76	74.92	89.04	88.08	90.32	75.56	73.18	90.62
	Fast1	7.94	11.32	11.28	7.60	7.52	8.12	9.68	10.06	8.76	9.62	10.04	8.76
	RMD	***1.61***	***1.40***	***1.45***	***1.61***	***1.61***	***1.61***	***1.61***	***1.59***	***1.65***	***1.55***	***1.52***	***1.65***
	Fast2	7.52	11.36	11.28	7.42	7.30	8.06	10.08	10.08	9.36	10.02	10.02	9.22
	RMD	***1.63***	***1.40***	***1.45***	***1.62***	***1.62***	***1.61***	***1.59***	***1.59***	***1.62***	***1.53***	***1.52***	***1.63***
30	Slow	1358.00	1190.00	1191.60	953.60	957.60	779.60	1354.20	1354.80	330.20	1342.88	1352.40	336.00
	Fast	3.20	4.40	4.40	2.00	2.00	2.00	4.80	4.40	2.60	4.00	4.00	2.60
	RMD	***1.99***	***1.99***	***1.99***	***1.99***	***1.99***	***1.99***	***1.99***	***1.99***	***1.97***	***1.99***	***1.99***	***1.97***
31	Slow	15.92	4.62	4.62	14.22	14.26	3.96	26.60	18.10	2.82	15.04	15.04	2.86
	Fast	3.48	4.60	4.60	3.12	3.12	2.98	4.60	5.12	1.76	4.44	4.60	1.74
	RMD	***1.28***	***0.00***	***0.00***	***1.28***	***1.28***	***0.28***	***1.41***	***1.12***	***0.46***	***1.09***	***1.06***	***0.49***
32	Slow	7.10	7.54	11.26	4.18	4.18	4.54	13.38	14.32	4.38	7.52	7.46	4.54
	Fast	3.40	4.60	4.62	3.12	3.16	2.96	4.60	4.74	2.50	4.50	4.60	2.90
	RMD	***0.70***	***0.48***	***0.84***	***0.29***	***0.28***	***0.42***	***0.98***	***1.01***	***0.55***	***0.50***	***0.47***	***0.44***
33	Slow	259.18	225.34	224.08	62.08	62.06	30.38	274.16	272.52	30.00	322.62	335.02	30.03

	Fast	18.34	21.14	21.32	14.86	14.86	10.62	23.16	23.36	11.88	23.96	23.84	11.96
	RMD	***1.74***	***1.66***	***1.65***	***1.23***	***1.23***	***0.96***	***1.69***	***1.68***	***0.87***	***1.72***	***1.73***	***0.86***
	Slow	486.04	414.22	414.38	334.86	333.30	241.96	553.10	556.12	262.20	560.28	559.46	275.64
34	Fast	214.34	156.22	147.32	207.18	207.76	126.64	181.76	181.94	131.24	192.52	185.76	127.26
	RMD	***0.78***	***0.90***	***0.95***	***0.47***	***0.46***	***0.63***	***1.01***	***1.01***	***0.67***	***0.98***	***1.00***	***0.74***
	Slow	4812.00	4260.00	4230.00	3272.00	3242.00	2438.00	5314.00	5424.00	1580.00	5384.00	5348.00	1616.00
35	Fast	526.00	530.00	530.00	364.00	360.00	302.00	644.00	646.00	202.00	620.00	634.00	206.00
	RMD	***1.61***	***1.56***	***1.55***	***1.60***	***1.60***	***1.56***	***1.57***	***1.57***	***1.55***	***1.59***	***1.58***	***1.55***
	Slow	1162.20	1042.00	1031.20	1052.60	1001.00	821.60	1407.00	1404.40	354.60	1455.00	1468.60	370.40
37	Fast	159.80	152.20	159.20	208.00	202.00	274.00	165.80	160.60	21.60	174.80	173.20	20.60
	RMD	***1.52***	***1.49***	***1.47***	***1.34***	***1.33***	***1.00***	***1.58***	***1.59***	***1.77***	***1.57***	***1.58***	***1.79***
	Slow	11928.00	9904.00	9084.00	12300.00	12228.00	7926.00	15054.00	1568.60	6346.00	13426.00	13214.00	6447.00
38	Fast	1422.00	1244.00	1236.00	1028.00	1024.00	856.00	1414.00	150.60	344.00	1416.00	1406.00	334.00
	RMD	***1.57***	***1.55***	***1.52***	***1.69***	***1.69***	***1.61***	***1.66***	***1.65***	***1.79***	***1.62***	***1.62***	***1.80***
	Slow	3.46	6.18	6.02	2.24	2.26	4.16	6.38	6.06	2.88	6.24	6.24	3.36
39	Fast	2.82	4.60	4.20	2.26	2.28	2.38	5.02	5.14	2.46	4.60	4.60	2.24
	RMD	***0.20***	***0.29***	***0.36***	***-0.01***	***-0.01***	***0.54***	***0.24***	***0.16***	***0.16***	***0.30***	***0.30***	***0.40***
	Slow	6.06	3.78	2.96	5.72	6.64	4.84	3.10	3.14	2.82	9.28	9.24	4.62
41	Fast	1.84	1.24	1.00	1.50	2.08	1.96	2.38	2.38	0.94	2.46	2.46	1.56
	RMD	***1.07***	***1.01***	***0.99***	***1.17***	***1.05***	***0.85***	***0.26***	***0.28***	***1.00***	***1.16***	***1.16***	***0.99***
	Slow	18.66	17.22	17.40	17.84	17.98	14.60	17.48	17.68	14.04	19.62	19.58	15.18
42	Fast	0.62	1.08	1.12	0.50	0.46	0.88	0.60	0.60	0.50	0.60	0.60	0.52
	RMD	***1.87***	***1.76***	***1.76***	***1.89***	***1.90***	***1.77***	***1.87***	***1.87***	***1.86***	***1.88***	***1.88***	***1.87***

4.4.3 Testing the Performance of Real Applications

To further illustrate the performance of our refactoring system, we randomly select a number of AS3 applications from the Internet[34,35]. We test the execution time of the original code before refactoring and the execution time of the refactored code after ART is utilized to parse and transform the original code. We measure the performance of the applications across the function being refactored because testing a system's execution time is not feasible. Interactive Flash applications including games usually do not have explicit ending points.

For example, we test the execution time of the function collisionCheckVertical() in the motivating example (Section 4.1). The test environment is Firefox 3.6 with Adobe Flash Player 10 installed. After applying ART to this function, the optimized ActionScript code is as follows.

```
function collisionCheckVertical():void {
  var length1:Number = body.block.length;
  for(ii = 0; ii < length1; ii++){
    if(body.yp + body.block[ii].yp >= cnvMain.height -
    BLOCKSIZE){
      commitBody();
      return;
    }
  }
  var length2:Number = aryBlockRow.length;
  for(jj = 0; jj < length2; jj++) {
    var length3:Number = aryBlockRow[jj].length;
    for(ii = 0; ii < length3; ii++){
      if(aryBlockRow[jj][ii]){
        if(body.CollisionCheck(aryBlockRow[jj][ii])){
          commitBody();
          return;
        }
      }
    }
  }
}
```

The pattern "Avoid array.length in for statements" (declare a variable to get the length of array outside the loop, and use the value inside the loop) is used three times to the collisionCheckVertical() function. After the transformation, we test the execution time of the optimized function collisionCheckVertical() and calculate the relative mean difference (0.40 RMD) of the the execution time before and after refactoring.
Table 4.2 shows the execution time (in milliseconds) of the original (slow), the refactored (fast) code and RMD for 9 functions in 9 Flash applications running in three different browsers. To test the execution time before and after refactoring, we develop the code in CS4, used Adobe Flash Player 9 and 10 as the add-ons to

[34] http://www.krazydad.com/bestiary/
[35] http://flash.9ria.com/

the runtime environments: Internet Explorer 8.0, Firefox 3.6 and Chrome 5.0. Figure 4.6 illustrates the value of the RMD of refactoring patterns in these 9 Flash applications tested in different browsers. Table 4.2 and Figure 4.6 clearly demonstrate the efficiency is significantly improved with ART. Table 4.2 and Figure 4.6 also demonstrate that different configurations have different impacts on the effectiveness of any refactoring approach. Hence, it is essential that Flash programmers understand the impact of their configuration selections if they are to produce highly efficient solutions.

According to the 80-20 rule (10% to 20% of the code occupies 80% to 90% of the execution time), it is only worthwhile to refactor the bottlenecks of a program, not every line of code. Thus, before refactoring, an internal ActionScript profiler is used (Adobe Flex Builder36) to analyze the performance of a program. Based on the performance analysis, we only refactor the function with the "largest" computational overhead. The improvements presented in Table 4.2 are the average across a large number of executions when the functions are supplied with random, but valid, inputs.

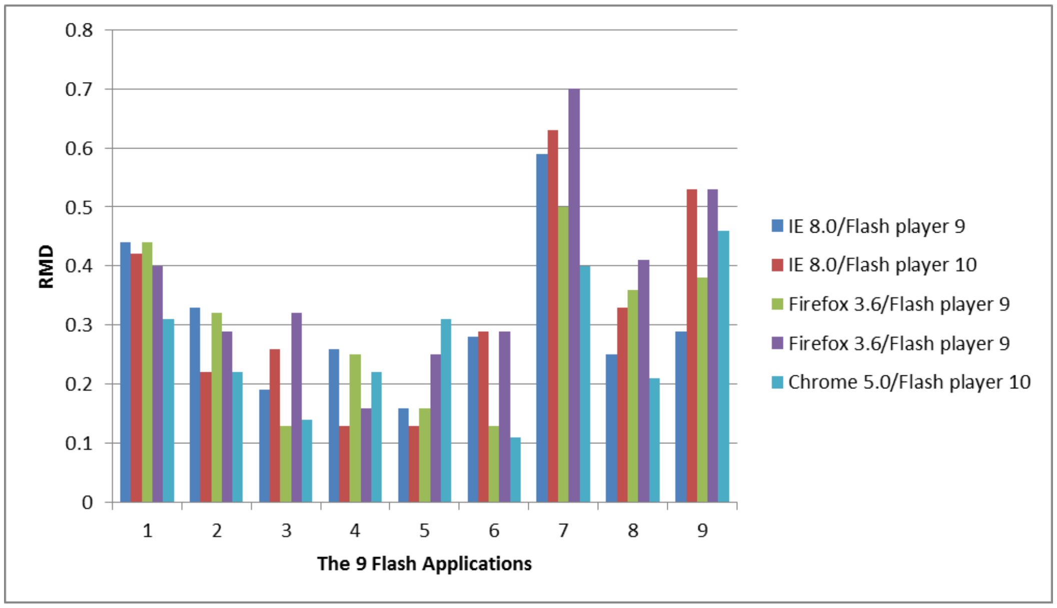


Figure 4.6 Improvement of Refactoring Patterns Tested in Different Browsers

[36] http://www.adobe.com/products/flex/

Table 4.2 Testing Results of Refactoring Patterns for Flash Applications

Application Name		Internet Explorer 8.0 (milliseconds) Adobe Flash Player 9	Internet Explorer 8.0 (milliseconds) Adoble Flash Player 10	Firefox 3.6 (milliseconds) Adobe Flash Player 9	Firefox 3.6 (milliseconds) Adoble Flash Player 10	Chrome 5.0 (milliseconds) Adobe Flash Player 10
Tetris	Before	9.4	6.4	9.4	6.0	3.0
	After	6.0	4.2	6.0	4.0	2.2
	RMD	***0.44***	***0.42***	***0.44***	***0.40***	***0.31***
Seek Road	Before	14.2	9.0	13.6	8.8	8.2
	After	10.2	7.2	9.8	6.6	6.6
	RMD	***0.33***	***0.22***	***0.32***	***0.29***	***0.22***
Object Cell	Before	20.6	9.0	19.4	8.8	6.2
	After	17.0	6.9	17.0	6.4	5.4
	RMD	***0.19***	***0.26***	***0.13***	***0.32***	***0.14***
Mine Sweeping	Before	20.8	8.0	20.8	8.0	5.0
	After	16.0	7.0	16.2	6.8	4.0
	RMD	***0.26***	***0.13***	***0.25***	***0.16***	***0.22***
Fern	Before	34.4	25.0	34.2	24.4	23.6
	After	29.2	22.0	29	19	17.2
	RMD	***0.16***	***0.13***	***0.16***	***0.25***	***0.31***
Bomb Pig	Before	74.4	8.0	68.6	8.0	10.0
	After	56.4	6.0	60.0	6.0	9.0
	RMD	***0.28***	***0.29***	***0.13***	***0.29***	***0.11***
Lightning	Before	7.0	5.0	7.0	5.8	3.6
	After	3.8	2.6	4.2	2.8	2.4
	RMD	***0.59***	***0.63***	***0.50***	***0.70***	***0.40***
Grass	Before	11.5	7.7	10.5	7.3	5.2
	After	8.9	5.5	7.3	4.8	4.2
	RMD	***0.25***	***0.33***	***0.36***	***0.41***	***0.21***
Supper Ball	Before	26.0	19.0	28.5	18.0	12.0
	After	19.5	11.0	19.5	10.5	7.5
	RMD	***0.29***	***0.53***	***0.38***	***0.53***	***0.46***

The RMD as shown in Table 4.2 are between 0.11 and 0.67. This is because the performance of the refactoring patterns varies across applications due to differing code structures. On one hand, the frequency of occurrence for a pattern influences the pattern's performance. For example, the assignment operator is faster than the push() method to set an array value (as mentioned in previous Section). However, if push() is only used once outside of a loop, the performance improvement is not obvious. On the other hand, the code that is unchanged by refactoring strongly affects the performance of our patterns. We use implicit path enumeration (Li & Malik, 1997) to illustrate. Let c_i be the execution time of the basic program block, x_i be the number of times the basic block is run and N be the total number of basic blocks, thus, the total execution time is:

$$\sum_{i=1}^{N} c_i x_i \qquad (2)$$

The total execution time is undecidable without the constraints on x_i: structural constraints, concluded from program control flow graphs; and functionality constraints, given the loop bound or other path information by users according to functionality of a program. If we divide the function into two blocks: B_1 which does not have our patterns, B_2 which includes our patterns. The total execution time of this function is: ($c_1 * x_1 + c_2 * x_2$), where c_1 and c_2 are the time for executing B_1 and B_2 once respectively. Looking at Figure 4.7(a) and Figure 4.7(b), they both have the pattern "Replace Type Number for iterations". To get an accurate relative pattern performance, what we required is only the execution time of block1 (B_1). Therefore, the block2 (B_2) should be empty or the code structure of block2 (B_2) should be extremely simple to not affect the results, as shown in Figure 4.7(a). However, when we evaluate the pattern performance on the applications, we are required to test more extended code blocks to guarantee the integrity of the code structure, as is demonstrated in Figure 4.7 (b).

(a)

B_1

```
for(var i:Number = 0 ; i <= 100; i++){
```

B_2

```
tmp=i;
}
```

(b)

B_1

```
for (var i:Number = 0; i <= 100; i++){
```

B_2

```
mc = new GrassBlade();
mc.x = i*5;
mc.y  =  itsPar.stage.stageHeight-50;
itsPar.addChild(mc);
}
```

Figure 4.7 The Effects of Code Structure on Refactoring Patterns

The total execution time for the code in Figure 4.7(a) and Figure 4.7(b) is: (c_1 * 100 + c_2 * 100), where c_1 and c_2 are the times for executing B_1 and B_2 once respectively. The influence of the "Replace Type Number for iterations" pattern is only to reduce the execution time of B_1. The influence of the refactoring is only to reduce the execution time of B_2. Therefore, if B_1 takes much more time than B_2, even though the patterns are inside a loop, the improvement of c_2 will not be obvious. Therefore, the performance of the refactoring patterns varies across applications due to differing code structures. Nevertheless, the results show a positive improvement, which means ART has the ability to speed up Flash applications by translating bad smell coding structures into more efficient code structures. (2) The performance of ART is dependent on how many patterns are inside a function; this reflects on the programmers' programming skills. More skillful programmers will have fewer patterns in their functions while less skillful programmers will have more patterns.

Chapter 5 Refactoring Flash Embedding Methods

Though both Flash and Ajax allow programmers to build dynamic websites, they have different focuses. Flash stresses high-quality graphics and animations. It supports sound effects, video and audio playback and capture; whereas Ajax emphasizes actions on a website that involves the web server and the browser. Integrating Flash with Ajax to enhance the user experience is highly popular. In practice, Flash is usually served as a partial substitute for the interface of Ajax technology to provide many graphical tasks that are difficult to accomplish using only Ajax. Currently, there are many innovative websites using Flash and Ajax technologies together, such as Google Finance[37] and Yahoo Finance[38].

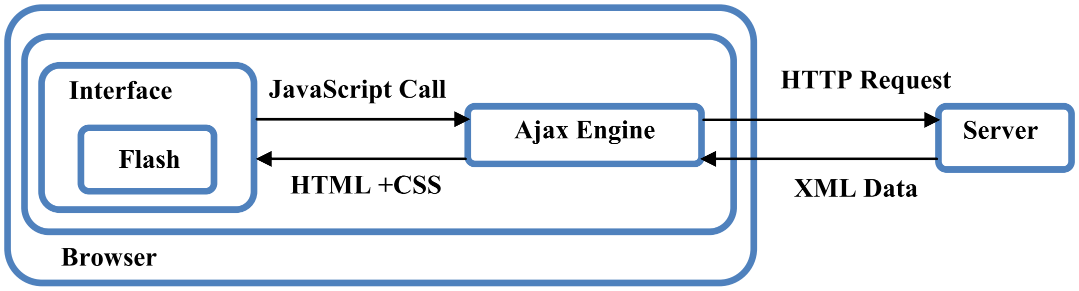


Figure 5.1 The Combined Ajax and Flash Working Model

Figure 5.1 shows the combined Flash and Ajax working model. Flash content is integrated into the user interface as a component. The Ajax engine runs within the browser to communicate, interact and display information between the server and the browser. If more data is requested from the Flash component, the Ajax engine sends an asynchronous request to the server to retrieve extra data for the Flash component to display without causing the entire web page to be refreshed.

Flash is written in ActionScript; whereas Ajax uses JavaScript. To implement the integration of Flash and Ajax technologies, interaction between ActionScript and JavaScript follows two steps.

1. Embedding Flash content into a web page.
2. Communication between ActionScript and JavaScript.

Flash can be embedded through markup-based embedding methods or JavaScript-based embedding methods (Starr, 2008). Markup-based embedding methods use pure Hypertext Markup Language (HTML) tags <object> or <embed> to include Flash content while JavaScript-based embedding methods utilize JavaScript to load Flash content. Flash embedding methods should meet the following criteria (Sluis, 2007; Braunstein et al., 2007).

1. Cross-browser support: the method should have the ability to support all web browsers.

[37] http://www.google.com/finance
[38] http://finance.yahoo.com/

2. Standards compliance: the method should be in compliance with the World Wide Web Consortium (W3C) standards[39].
3. Support for alternative content: alternative content should be supported. That is, the content should be accessible when no Adobe Flash Player is installed. This content can also be used for search engine indexing purposes.
4. Support for plug-in version detection: The version of the Adobe Flash Player should be detected before the Flash content is displayed. Mismatches between the Flash content and the Adobe Flash Player may result in errors or broken content.

In general, markup-based Flash embedding methods provide no Flash content and plug-in version detection; whereas JavaScript-based Flash embedding methods meet all the criteria (the detail will be discussed in the Markup-based Flash Embedding Method and JavaScript-based Flash Embedding Methods Sections). Even with limitations with markup-based methods, most programmers still use them to embed Flash content (Schmitt, 2005). This is because: (1) many programmers have limited knowledge about the pros and cons of markup-based and JavaScript-based Flash embedding methods. (2) Programmers are more familiar with the <object> and <embed> tags than the new JavaScript libraries. Although most of the JavaScript libraries for embedding Flash content are easy to use, it takes time for programmers to learn and become familiarize with the libraries. (3) Most Flash embedding tutorials and Flash publishing tools choose to use mark-up based methods for simplicity purposes, and programmers following the tutorials or using Flash publishing tools just accept the default without learning more about JavaScript-based methods.

Due to the disadvantages of markup-based embedding methods, they should be replaced by JavaScript-based embedding methods. Clearly, a manual transformation that requires the programmer to be knowledgeable about the HTML tags (the <object> and <embed> tag) and the JavaScript library can introduce defects. To aid programmers with the transformation from markup-based embedding methods to JavaScript-based embedding methods, we have built a refactoring tool, FlashembedRT. This tool refactors markup-based embedding methods into a method using one of the popular Flash embedding JavaScript libraries, flashembed[40].

This chapter is organized as follows: Section 5.1 and Section 5.2 introduce existing markup-based and JavaScript-based Flash embedding methods; Section 5.3 discusses our refactoring process.

[39] http://www.w3.org/
[40] http://flowplayer.org/tools/toolbox/flashembed.html

5.1 Markup-based Flash Embedding Methods

Five different markup-based embedding methods are frequently used to insert Flash content into a web page. To explain the differences between the methods, we provide an example configuration for embedding Flash content (Table 5.1).

Table 5.1 An Example Configuration

Attribute	Value
Container	<div id= “flash” >
Path	path/flash_movie.swf
Width	300
Height	300
alternative content	<p>Alternative content</p>
required Adobe Flash Player version	9.0.45.0
flashvars1	name = varialbe1; value = value1
flashvars2	name = varialbe2; value = value2

For each method, we provide the HTML code using the example configuration as well as the grammar for the HTML code. We have extended the HTML grammar[41] and to make the grammar simpler, only the directly utilized symbols and rules of the grammar are included.

5.1.1 The <embed> tag

Using the <embed> tag is the most convenient way to insert Flash content. All major browsers support this method; however, it is not standards-compliant. The <embed> tag is invalid in HTML 4 and XHTML 1. Though this method supports alternative content using <noembed> tag, it fails to detect the version of Adobe Flash Player. The Flash embedding code for the <embed> tag is as follows.

```
<div id = "flash">
   <embed type = "application/x-shockwave-flash"
     pluginspage = "http://www.adobe.com/go/getflashplayer"
     width = "300" height = "300" src = "path/flash_movie.swf"
     flashvars = "variable1=value1&variable2=value2"/>
   <noembed><p>Alternative content</p></noembed>
</div>
```

This code segment can be explained as follows.

1. The type attribute specifies that the embedded content is Flash content.
2. The pluginspage attribute indicates the location of the Adobe Flash Player.
3. The width and height attributes are required attributes of the <embed> tag to specify the dimensions of the Flash content.
4. The src attribute specifies the location of the Flash content and the flashvars attribute defines variables to be passed to the Adobe Flash Player.

[41] http://www.antlr.org/grammar/HTML/html.g

The HTML grammar for this method is defined as follows.

```
div: '<div' (WS ATTR)? '>' (body_content)* '</div>';
body_content: body_tag | text;
body_tag: heading | block | ADDRESS;
text:  PCDATA | text_tag;
text_tag: font | phrase | special | form;
special: embed | noembed | ((condition)* object (condition)*)) |
anchor | IMG | applet | font_dfn | BFONT | map | BR;
embed:'<embed'  WS  ('type  =  "application/x-shockwave-flash"')
('pluginspage  =  "http://www.adobe.com/go/getflashplayer"')  ('src
= "' (WORD ('%')? | ('-')? INT | STRING | HEXNUM) '"') ('width =
"' INT '"') ('height = "' INT '"') (ATTR)* '</>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
noembed: '<noembed>' (body_content)* '</noembed>';
```

5.1.2 The <object> tag

The <object> tag is recommended by the W3C to embed Flash content into a web page. Thus, this method is standards-compliant and alternative content is allowed. However, the <object> tag does not support plug-in detection functionality; and not all major browsers support it.

To insert Flash content into non-IE browsers, the MIME-type (flash) is specified for the type attribute. The data attribute indicates the location of the Flash content; the width and height attributes are required attributes of the <object> tag to indicate the dimensions of the Flash content and the <param> tag defines variables to be passed to the Adobe Flash Player using the flashvars attribute. The HTML code for non-IE browsers is as follows.

```
<div id = "flash">
  <object type = "application/x-shockwave-flash" width = "300"
    height = "300" data = "path/flash_movie.swf">
    <param name = "flashvars"
      value = "variable1=value1&variable2=value2"/>
   <p>Alternative content</p>
  </object>
</div>
```

To insert Flash content into IE, the classid attribute is used to identify the ActiveX control for the browser as IE expects the Adobe Flash Player to be an ActiveX control. The movie attribute for the <param> tag specifies the location of the Flash content. The HTML code for IE is as follows.

```
<div id = "flash">
  <object classid = "clsid:D27CDB6E-AE6D-11cf-96B8-44455354000"
    width = "300" height = "300">
    <param name = "movie" value="path/flash_movie.swf"/>
    <param name = "flashvars"
      value = "variable1=value1&variable2=value2"/>
    <p>Alternative content</p>
  </object>
</div>
```

The HTML grammar for the <object> tag is as follows.

```
object: '<object' WS ('classid = "clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000"') | (('type = "application/x-shockwave-flash"')
('data = "' (WORD ('%')? | ('-')? INT | STRING | HEXNUM) '"'))
('width = "' INT '"') ('height = "' INT '"') (ATTR)* '>' (param)*
(body_content)* '</object>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
param: '<param name = "' WORD '"' 'value = "' (WORD ('%')? | ('-')?
INT | STRING | HEXNUM) '"</>';
```

5.1.3 Twice Cooked

The Twice Cooked method utilizes a nested <embed> tag inside an <object> tag to embed Flash content. The method is widely used because it is the default method for the Adobe Flash IDE to insert Flash content. Nonetheless, it is redundant as each value is declared twice; it is not standards-compliant due to usage of the <embed> tag; and it provides no alternative content and plug-in version detection functionality. The codebase attribute specifies the download location of the Adobe Flash Player. When no Adobe Flash Player is installed, the browser downloads the Adobe Flash Player automatically. The HTML code for this method is as follows.

```
<div id = "flash">
  <object classid = "clsid:D27CDB6E-AE6D-11cf-96B8-44455354000"
    codebase = "http://fpdownload.macromedia.com/pub/shockwave/
    cabs/flash/swflash.cab#version=9,0,45,0" width = "300"
    height = "300">
    <param name = "movie" value = "path/flash_movie.swf"/>
    <param name = "flashvars"
      value = "variable1=value1&variable2=value2"/>
    <embed type = "application/x-shockwave-flash"
      src = "path/flash_movie.swf" width = "300" height = "300"
      pluginspage = "http://www.adobe.com/go/getflashplayer"
      flashvars = "variable1=value1&variable2=value2"/>
  </object>
</div>
```

The HTML grammar for this method is as follows.

```
object: '<object' WS ('classid = "clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000"') ('codebase = " http://fpdownload.macromedia.com/
pub/shockwave/cabs/flash/swflash.cab#version='    version    '"')
('width = "' INT '"') ('height = "' INT '"') (ATTR)* '>' (param)+
embed '</object>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
param: '<param name = "' WORD '"' 'value = "' (WORD ('%')? | ('-')?
INT | STRING | HEXNUM) '"</>';
embed: '<embed' WS ('type = "application/x-shockwave-flash"')
('pluginspage = "http://www.adobe.com/go/getflashplayer"') ('src
= "' (WORD ('%')? | ('-')? INT | STRING | HEXNUM) '"') ('width =
"' INT '"') ('height = "' INT '"') (ATTR)* '</>';
```

5.1.4 Nested Objects

Instead of using a nested <embed> tag, the Nested Objects method utilizes a nested <object> tag to embed Flash content. It is standards-compliant and supports alternative content.

```
<div id = "flash">
  <object classid = "clsid:D27CDB6E-AE6D-11cf-96B8-44455354000"
    width = "300" height = "300">
    <param name = "movie" value = "path/flash_movie.swf"/>
    <param name = "flashvars"
      value = "variable1=value1&variable2=value2"/>
    <object type = "application/x-shockwave-flash"
      data = "path/flash_movie.swf" width = "300"
      height = "300">
      <param name = "flashvars"
        value = "variable1=value1&variable2=value2"/>
      <p>Alternative content</p>
    </object>
  </object>
</div>
```

The Nested Objects method does not allow plug-in detection and lacks cross-browser support (Sluis, 2008). Using IE-specific conditional comments can solve the cross-browser problem (Murphy & Persson, 2008; Allsopp, 2009). IE-specific conditional comments provide a mechanism to target blocks of HTML code toward a specific version of the browser. It starts with a <!--[if]> tag and ends with a <![endif]--> tag.

```
<div id = "flash">
  <object classid = "clsid:D27CDB6E-AE6D-11cf-96B8-44455354000"
    width = "300" height = "300">
    <param name = "movie" value = "path/flash_movie.swf"/>
    <param name = "flashvars"
      value = "variable1=value1&variable2=value2"/>
    <!--[if !IE]>-->
    <object type = "application/x-shockwave-flash"
      data = "path/flash_movie.swf" width = "300"
      height = "300">
      <param name = "flashvars"
        value = "variable1=value1&variable2=value2"/>
    <!--<![endif]-->
    <p>Alternative content</p>
    <!--[if !IE]>-->
    </object>
    <!--<![endif]-->
  </object>
</div>
```

The conditional comments [if !IE] targets non-IE browsers, so that all browsers (IE and non-IE) are considered by this method. The HTML statements inside the conditional comments are evaluated when non-IE browsers are detected. This method is standards-compliant and it provides support for all major browsers. However, it is redundant and it has no plug-in detection functionality. Another problem with this method is that IE cannot stream large movies; the movie only starts playing after the entire video file is downloaded. The HTML grammar for this method is as follows.

```
object: '<object' WS ('classid = "clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000"') | (('type = "application/x-shockwave-flash"')
('data = "' (WORD ('%')? | ('-')? INT | STRING | HEXNUM) '"'))
('width = "' INT '"') ('height = "' INT '"') (ATTR)* '>' (param)*
(condition)* (body_content)* (condition)* '</object>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
param: '<param name = "' WORD '"' 'value = "' (WORD ('%')? | ('-')?
INT | STRING | HEXNUM) '"</>';
```

5.1.5 Flash Satay

The Satay method (Mclellan, 2002) is based upon a generic object implementation and can solve the streaming problem of Internet Explorer. A small container is utilized to load the Flash content. The implementation of this method contains two steps: (1) create a new Flash movie called c.swf and place _root.loadMovie(_root.path,0) into the first frame. (2) The actual Flash movie is loaded using the following code, which passes a variable (path) to the c.swf to load the target flash_movie.swf.

```
<div id = "flash">
  <object type = "application/x-shockwave-flash"
     data = "c.swf?path=movie.swf" width = "300" height = "300">
    <param name = "movie"
      value = "c.swf?path=path/flash_movie.swf"/>
    <param name = "flashvars"
      value = "variable1=value1&variable2=value2"/>
    <p>Alternative content</p>
  </object>
</div>
```

The HTML grammar for this method is as follows.

```
object:'<object' WS ('type = "application/x-shockwave-flash"')
('data = "' (WORD ('%')? | ('-')? INT | STRING | HEXNUM)) ('width
= "' INT '"') ('height = "' INT '"') (ATTR)* '>' (param)+
(body_content)* '</object>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
param: '<param name = "' WORD '"' 'value = "' (WORD ('%')? | ('-')?
INT | STRING | HEXNUM) '"</>';
```

5.2 JavaScript-based Flash Embedding Methods

To avoid the issues associated with markup-based Flash embedding methods, JavaScript-based Flash embedding methods have been developed to meet all the criteria for embedding Flash content. JavaScript-based Flash embedding methods can be implemented using different JavaScript libraries, such as SWFObject[42].

[42] http://code.google.com/p/swfobject/

Flashembed is a new JavaScript tool to embed Flash content into a web page, which has the following features.

1. Flash can be configured with JSON. This is a unique feature of flashembed. This feature allows more complex configurations to be used when embedding Flash content.
2. Flashembed has jQuery support; it can work as a standalone tool as well as a jQuery plug-in.
3. Flashembed produces standards-compliant markup, provides plug-in detection and alternative content, and is supported by all the major browsers.

Flashembed is easy to use; the syntax for the flashembed function is as follows.

```
flashembed(container, embedOptions, flashConfiguration);
```

The container argument indicates which HTML element contains the Flash object; the embedOptions argument specifies the path to the swf file and all the attributes for embedding; and the flashConfiguration argument configures the Flash object by providing flashvars to the Flash object. JSON-based configurations are allowed in the third argument.

The jQuery syntax for flashembed is shown as follows.

```
$("jquery_selector").flashembed(embedOptions, flashConfiguration);
```

Flashembed provides plug-in detection and alternative content in the following ways.

1. Programmers place alternative content (HTML code) directly into the HTML element where Flash is embedded.
2. Flashembed has an attribute called version, which indicates the required version of the Adobe Flash Player to display the Flash content. By specifying this attribute, Flashembed detects the version of the Adobe Flash Player upon loading the Flash content. If the required version is not detected, the default alternative content will be delivered to the user.
3. Adobe's Express Install (Kazoun & Lott, 2008; Carey, 2009) is supported through the expressInstall property. Express Install detects the version of the Adobe Flash Player, if it is not the latest version; it allows a process to update the Adobe Flash Player to the latest version.
4. Flashembed provides an onFail method which is evaluated when no Adobe Flash Player or an old version of the Adobe Flash Player is detected.

5.3 Refactoring Flash Embedding Methods

We adopt refactoring for migration purpose. The semi-automatic refactoring approach (Mens & Tourwé, 2004) is used as refactoring will often encompass functionality which cannot be inferred automatically by a program. Thus, programmers are required to provide information to accomplish the transformation. The invariants, pre and post-conditions refactoring technique (Opdyke, 1992; Roberts, 1999) is employed to ensure that the refactoring can be implemented successfully.

Pre-condition: The syntax of the mark-up based Flash embedding methods is functionally correct.

Post-condition: The syntax of the generated JavaScript for the flashembed function and the HTML code for the alternative content is functionally correct.

Our refactoring process comprises three phases: bad smell detection, code rewriting and code generation.

5.3.1 Bad Smell Detection

The first step of our transformation process is to detect bad smells. The bad smell in our system comes from the five different markup-based Flash embedding methods. These methods utilize the HTML <object> and <embed> tags to include Flash content, which can be identified using an HTML parser. For this dissertation, Jsoup[43], a Java HTML parser, is adopted to find, extract and modify HTML elements, attributes and the content of the elements. Bad smell detection includes:

1. Flash Container Extraction

When embedding Flash content, it must be placed into an HTML element such as the <div> tag. The first argument of the flashembed function indicates the id of the HTML element where the Flash object is placed. Thus, the Flash container needs to be extracted.

2. Attribute Extraction

Flash supports different parameters or attributes in the <object> or <embed> tag to control how the Flash content is embedded (Carey, 2009). When using the <object> tag to embed Flash content, programmers are required to define the attributes for both of the <object> tag and the nested <param> tag. The flashembed function has one required attribute, src, and several optional attributes that are specific to the function.

- The version attribute specifies the minimum version of the Adobe Flash Player required for displaying Flash content. The format for the version number is [major, fix].
- The w3c attribute specifies whether the HTML code using standards-based syntax is generated to include Flash content. If the w3c attribute is set to false, flashembed generates different HTML code using the <object> tag according to the browser type (IE or non-IE). By enabling this attribute, flashembed generates a standards-based syntax that supports all browsers.
- The cachebusting attribute decides whether flashembed forces Flash content to be loaded from the server while ignoring the Expires HTTP header of the

[43] http://jsoup.org/

browser's cache.

- The expressInstall attribute specifies an absolute or relative path to an expressinstall.swf file which allows users to update the Adobe Flash Player to the latest version.

The flashembed function also allows the usage of popular attributes associated with the <object> and <embed> tags, such as bgcolor, wmode, allowfullscreen and allowscriptaccess. Thus, attribute extraction extracts the <object> and <embed> tags' attributes that can be mapped to the corresponding attributes in the flashembed function, while ignoring specific attributes for the <object> and <embed> tag, such as the classid attribute for the <object> tag, and the pluginspage attribute for the <embed> tag.

3. Alternative Content Extraction

In markup-based Flash embedding methods, the alternative content is specified inside the <object> or <noembed> tag. If programmers select to keep the alternative content specified in markup-based Flash embedding methods, the HTML code for alternative content – which are to be preserved during the process of refactoring – is extracted.

4. Flash Variables Extraction

The <object> and the <embed> tags are able to pass variables from the HTML document to the Flash content through the flashvars attributes/parameters when the Flash content is loaded in a web browser. The variables are defined using a set of "name = value" pairs separated by the "&" character; whereas the third argument of the flashembed function defines flashvars using a set of "name: value" pairs separated by commas. Therefore, if the flashvars property is specified in the markup-based Flash embedding method, the name and value of the variable to be passed to the Flash content are extracted and transformed to the corresponding flashembed format.

5.3.2 HTML Code Rewriting

Moving from markup-based Flash embedding methods to JavaScript-based Flash embedding methods requires deleting the <object> and <embed> tags and their specifications. Only the alternative content in the containing element remains intact, if specified to be preserved. The HTML code rewriting phase cleans up the HTML code for markup-based Flash embedding methods.

5.3.3 Flashembed Code Generation

The flashembed code generation phase consists of six steps.

1. Select the style for the generated code

The first step to produce the flashembed function code is for programmers to select the style of the generated code: JavaScript or jQuery.

2. Code generation for importing external JavaScript libraries

The code for importing the flashembed library is produced in this step. If programmers decide to generate the flashembed function using jQuery, the code for importing the jQuery library is also produced.

3. Code generation for attributes

In this step, the attributes for embedding Flash content – extracted in the bad smell detection phase – are mapped to the attributes in the flashembed function. Additionally, flashembed has several specific attributes, such as w3c and cachebusting that require programmers to select and configure. The system will produce the code for all the extracted and newly selected attributes according to the syntax of the flashembed function.

4. Code generation for alternative content

If the alternative content specified in markup-based Flash embedding methods is required to be preserved, the alternative content is unchanged before and after refactoring. Flashembed provides programmers additional options for the Adobe Flash Player version detection and alternative content through the version, expressInstall properties and the onFail method. If programmers select to utilize the version property, they are required to provide the version number of the Adobe Flash Player to generate the code for this property. By specifying the version attribute, the default HTML snippet for alternative content is created. If the expressInstall property is selected, programmers need to input the path to the Adobe Flash Player express install for the code generation of this property. If programmers wish to add dynamic alternative content using JavaScript through the onFail method, a skeleton of the method is generated and programmers are required to add the body of the method as the system has no way of understanding the domain of the application.

5. Code Generation for Flashvars

If, during the bad smell detection phase, variables being passed to the Flash content are detected, the format of these variables are transformed into flashembded's format. This step produces the code for the third argument of the flashembed function.

6. Code Generation for the flashembed function

Once steps 1 to 5 are completed, the flashembed function is generated.

5.3.4 Grammar for Transformation

In the previous section, we provide the grammar for five different markup-based Flash embedding methods before refactoring. In this section, we provide the grammar for the flashembed function after refactoring. We extended JavaScript grammar[44]. To make the grammar simpler, only the directly utilized symbols and rules of the grammar are included. The grammar for JavaScript flashembed function is shown as follows.

[44] http://www.antlr.org/grammar/1206736738015/JavaScript.g

```
flashembed:   'flashembed('   container   ','   embedOptions   ','
flashConfiguration ')';
container: '"' Identifier '"';
embedOptions: '{' ('"' Identifier '"') | (attributes) '}';
attributes: attribute (',' attribute)*;
attribute: Identifier ':' ((('"')? Identifier ('"')?) | function);
flashConfiguration: '{' flashvar | JSONConfiguration '}';
flashvar: var (',' var)*;
var: Identifier ': ' ''' Identifier ''';
JSONConfiguration: Identifier ':{' members? '}'
members: pair (',' pair)*;
pair: Identifier ':' (''')? value (''')?;
value: Identifier | JSONConfiguration;
```

This is the grammar for the flashembed function using jQuery.

```
jquery:  '$('  jquery_selector  ').flashembed('  embedOptions  ','
flashConfiguration ')';
container: '"' Identifier '"';
embedOptions: '{' ('"' Identifier '"') | (attributes) '}';
attributes: attribute (',' attribute)*;
attribute: Identifier ':' ((('"')? Identifier ('"')?) | function);
flashConfiguration: '{' flashvar | JSONConfiguration '}';
flashvar: var (',' var)*;
var: Identifier ': ' ''' Identifier ''';
JSONConfiguration: Identifier ':{' members? '}'
members: pair (',' pair)*;
pair: Identifier ':' (''')? value (''')?;
value: Identifier | JSONConfiguration;
```

5.3.5 System in Operation

To illustrate the operation of FlashembedRT, we provide an example to demonstrate our refactoring process from the Nested Object method to the flashembed method. The system starts with the bad smell detection phase.

1. The id ("flash") of the Flash container (the <div> tag) is retrieved.
2. The width and height attributes of the parent <object> tag and the movie attribute of the nested <param> tag are extracted. The specifications for the attributes of the parent <object> and the nested <object> tag should be identical (except the type and the classid attributes). It is not necessary to extract all the attributes in the nested <object> tag. However, FlashembedRT extracts the attributes of the nested <object > tag and compares them with the parent <object> tag. If the specifications are different, an error will occur.
3. The alternative content, <p>Alternative content</p>, is retrieved.
4. The flashvars attribute is extracted and transformed to the corresponding format in the flashembed function.

The HTML code rewriting phase: we select to keep the alternative content. Hence, the parent <object> tag and the nested <object> tag are deleted, only the alternative content is preserved in the HTML code.

The flashembed code generation phase:

1. We select to generate the flashembed function in JavaScript.
2. The code for importing the flashembed library is produced.
3. We specify the w3c attribute to be true, and then the code for the w3c attribute and other attributes are generated. The movie attribute of the nested <param> tag is mapped to the src attribute of the flashembed function.
4. To add Adobe Flash Player version detection and alternative content, we specify the version attribute to be 9.0.45.0. By specifying the version attribute, the default HTML snippet for alternative content is created in the container of the Flash content. However, in our example, the alternative content have already defined in the Flash container; hence, the default HTML code will be disabled. We also select to use the onFail method, so the code for the version attribute and the skeleton of the onFail method are produced.
5. The code for the flashvars attribute is generated.
6. The flashembed function is produced according to our configuration and the last step is to add JavaScript code to the onFail function. We add JavaScript code to change the title of the web page if no Adobe Flash Player or an old version of the Adobe Flash Player is detected.

After FlashembedRT is applied to the Nested Objects method, the generated code for the flashembed function is as follows.

```
flashembed("flash", {
  src: "path/flash_movie.swf",
  width: "300",
  height: "300",
  version: [9, 45],
  onFail: function() {
    document.title = "Get Adobe Flash Player";
  },
  variable1: 'value1',
  variable2: 'value2'
});
```

Chapter 6 Refactoring to Switch the Data Exchange Language for Improving Ajax Application Performance

To achieve a more responsive user experience, data transmission rates and performance (on both the client and server side) characteristics for Ajax applications are quite crucial. XML and JSON are two common data interchange formats currently used in Ajax applications. Though XML is a standard way to exchange data, due to its verbose nature, it is often replaced by JSON which is a lightweight data exchange format. Changing the data format from using XML to JSON for Ajax applications can improve the efficiency of an Ajax application (up to one hundred times faster) and enhance user experience by reducing the network transfer time and client JavaScript processing time (Nurseitov et al., 2009).

One way to implement this transformation is through refactoring. Our refactoring process is for efficiency purpose. Programmers can use our proposed refactoring technique to transform existing XML-based Ajax web applications into JSON-based Ajax web applications to increase the efficiency of their applications. To avoid tedious, error-prone and omission-prone (Dig et al., 2009a) manual refactoring, in this chapter, we introduce a refactoring tool, XtoJ, to aid with the transformation process. Specifically, the system assists programmers who maintain RIAs with the refactoring of systems that are still using XML data structures into JSON data structures.

Although JSON has been gaining in popularity with new Ajax applications, XML still remains very common. According to ProgrammableWeb.com[45], which is a website specializing in the categorization of Web APIs, there were 58% more XML-based Web APIs than JSON-based Web APIs as of August 2011. Additionally, many popular Web APIs from corporations such as Yahoo and Google initially only offered XML-based APIs before releasing JSON-based APIs at a much later date. Hence, Ajax web applications developed using XML-based APIs can now be refactored to take advantage of the available JSON-based APIs. As the popularity of JSON-based APIs increases, it is important for programmers to refactor their Ajax web applications from XML-based APIs because Web 2.0 websites may stop supporting XML-based APIs in the future. For example, Twitter stopped supporting XML-based Streaming API as of December 2010 (Irani, 2010).

The remainder of the chapter is as follows. Section 6.1 presents our refactoring approach to transform XML-based Ajax applications to JSON-based Ajax applications. Section 6.2 describes the three components of our refactoring system. Section 6.3 evaluates the performance of our refactoring system by utilizing it to refactor a number of real-world applications.

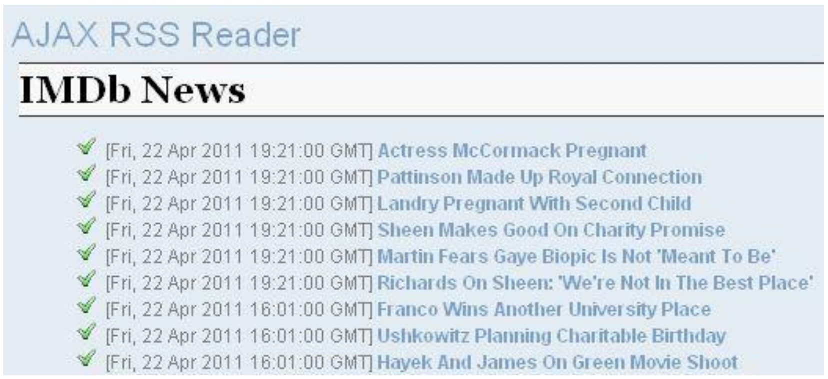
[45] http://www.programmableweb.com/

6.1 Refactoring XML into JSON in Ajax Applications

In Section 6.1.1, we provide an example of an XML-based Ajax application which will be used to demonstrate our refactoring process. We will then discuss the performance comparison between utilizing XML and JSON in Section 6.1.2. Finally, Section 6.1.3 describes our approach to implement the transformation from XML-based Ajax applications to JSON-based Ajax applications.

6.1.1 Example

We use the Ajax RSS Reader[46] and the Really Simple Syndication (RSS) XML file from IMDb[47] as an example. Figure 6.1 shows the web page of the Ajax RSS Reader (index.html). It displays the information of the channel and each feed in the channel.



AJAX RSS Reader

IMDb News

- [Fri, 22 Apr 2011 19:21:00 GMT] Actress McCormack Pregnant
- [Fri, 22 Apr 2011 19:21:00 GMT] Pattinson Made Up Royal Connection
- [Fri, 22 Apr 2011 19:21:00 GMT] Landry Pregnant With Second Child
- [Fri, 22 Apr 2011 19:21:00 GMT] Sheen Makes Good On Charity Promise
- [Fri, 22 Apr 2011 19:21:00 GMT] Martin Fears Gaye Biopic Is Not 'Meant To Be'
- [Fri, 22 Apr 2011 19:21:00 GMT] Richards On Sheen: 'We're Not In The Best Place'
- [Fri, 22 Apr 2011 16:01:00 GMT] Franco Wins Another University Place
- [Fri, 22 Apr 2011 16:01:00 GMT] Ushkowitz Planning Charitable Birthday
- [Fri, 22 Apr 2011 16:01:00 GMT] Hayek And James On Green Movie Shoot

Figure 6.1 The Ajax RSS Reader Web Page

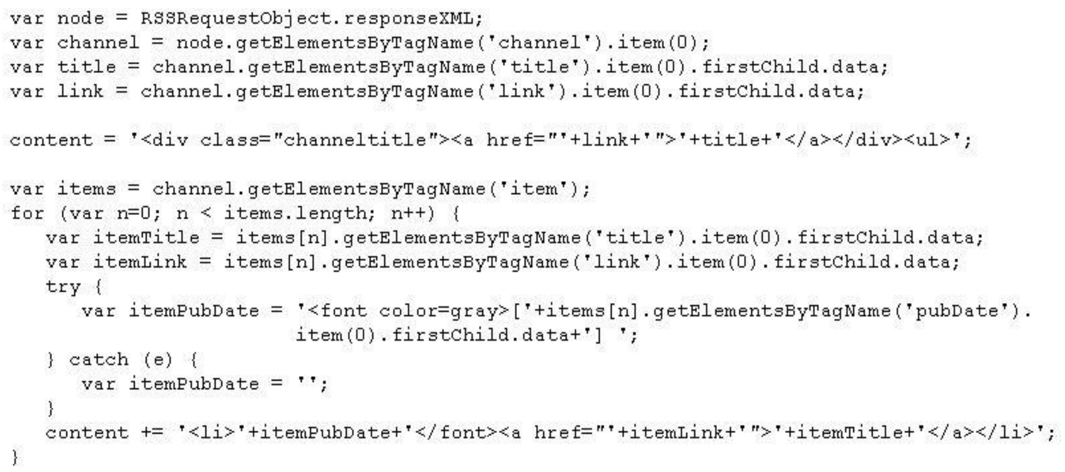
The structure of the RSS XML file is shown as follows.

```
<channel>
  <title>IMDb News</title>
  <link>http://www.imdb.com/rg/rss/news-channel/news/</link>
  <description>IMDb News</description>
  <language>en</language>
  <copyright>Copyright (C) 2011 IMDb.com, Inc.
    http://www.imdb.com/conditions</copyright>
  <image>
    <title>IMDb News</title>
    <url>http://i.imdb.com/logo.gif</url>
    <link>http://www.imdb.com/rg/rss/news-channel/news/
    </link>
  </image>
  <item>
    <title>Actress McCormack Pregnant</title>
    <pubDate>Fri, 22 Apr 2011 19:21:00 GMT</pubDate>
    <link>http://www.imdb.com/rg/rss/news/news/ni9872103/
    </link>
  </item>
```

46 http://ajax.phpmagazine.net/2005/11/ajax_rss_reader_step_by_step_t.html
47 http://www.imdb.com/

```
  <item>
    <title>Pattinson Made Up Royal Connection</title>
    <pubDate>Fri, 22 Apr 2011 19:21:00 GMT</pubDate>
    <link>http://www.imdb.com/rg/rss/news/news/ni9872102/
    </link>
  </item>
  …
</channel>
```

Figure 6.2 shows the JavaScript code used to retrieve data from the RSS XML file. In Figure 6.2, the responseXML property of the XMLHttpRequest object (RSSRequestObject) gets the response (XML) from a server and returns an XML DOM object (a tree structure). To get the value of the nodes, two XML DOM properties: firstChild (returns the first child of a node) and data (returns the value of a node), and one XML DOM method: getElementsByTagName(tagName) (returns all nodes with a specified tag name) are used.



```
var node = RSSRequestObject.responseXML;
var channel = node.getElementsByTagName('channel').item(0);
var title = channel.getElementsByTagName('title').item(0).firstChild.data;
var link = channel.getElementsByTagName('link').item(0).firstChild.data;

content = '<div class="channeltitle"><a href="'+link+'">'+title+'</a></div><ul>';

var items = channel.getElementsByTagName('item');
for (var n=0; n < items.length; n++) {
   var itemTitle = items[n].getElementsByTagName('title').item(0).firstChild.data;
   var itemLink = items[n].getElementsByTagName('link').item(0).firstChild.data;
   try {
      var itemPubDate = '<font color=gray>['+items[n].getElementsByTagName('pubDate').
                        item(0).firstChild.data+'] ';
   } catch (e) {
      var itemPubDate = '';
   }
   content += '<li>'+itemPubDate+'</font><a href="'+itemLink+'">'+itemTitle+'</a></li>';
}
```

Figure 6.2 The JavaScript Code for XML-based Ajax RSS Reader

6.1.2 Performance Comparison

The performance differences between utilizing XML and JSON is obvious. Nurseitov et al. (2009) design and implement scenarios to measure and compare the transmission time and resource utilizations between XML and JSON. They utilized massive data sets during this exploration. In the first scenario, a client sends 1,000,000 objects to a server using XML and JSON encoding; and in the second scenario, a client sends a series of smaller number of objects (20000, 40000, 60000, 80000 and 100000) to a server using XML and JSON encoding separately.

Table 6.1 Results for Different Scenarios

Scenario	Measurement	XML(ms)	JSON(ms)	RMD
Scenario 1	Total Time	4546694.78	78257.9	1.93
	Average Time Per Object	4.55	0.08	
Scenario 2	Total Time for 20,000 Objects	61333.68	2213.15	1.86
	Average Time Per Object	3.07	0.11	
	Total Time for 40,000 Objects	123854.59	3127.99	1.90
	Average Time Per Object	3.10	0.08	
	Total Time for 60,000 Objects	185936.27	4552.38	1.90
	Average Time Per Object	3.10	0.08	
	Total Time for 80,000 Objects	247639.81	6006.72	1.90
	Average Time Per Object	3.10	0.08	
	Total Time for 100,000 Objects	310017.47	7497.36	1.91
	Average Time Per Object	3.10	0.07	

Table 6.1 describes the performance differences between XML and JSON for these two scenarios. To compare their performances, the RMD of the transmission time of accessing XML data and JSON data is calculated as:

$$\frac{\textit{(Response Time for XML - Response Time for JSON)}}{\textit{(Response Time for XML + Response Time for JSON) /2}} \qquad (3)$$

It can be seen from the table that sending JSON encoded data is much faster than sending XML encoded data. This is because JSON is more compact than XML, which results in smaller documents. Thus, less bandwidth will be consumed and the data transmission between the client and server side will be significantly faster (Nurseitov et al., 2009). On the other hand, XML documents are usually accessed, and manipulated, by constructing an XML DOM (Jacobs, 2006). According to the definition by W3C, "The W3C Document Object Model (DOM)[48] is a platform and language-neutral interface (API) that allows programs and scripts to dynamically access and update the content, structure, and style of a document". To facility the node navigation, a DOM-based parser reads the entire XML document and transforms the XML document or XML string into an XML DOM object (a tree structure) in memory. When parsing large XML documents, it can be slow and resource-intensive (Jacobs, 2006).

However, JSON is a subset of JavaScript, retrieving data from a JSON object is "identical" to retrieving information from any other JavaScript object. Hence, processing JSON encoded data is much faster than processing XML encoded data, which makes the browser's response faster.

Therefore, based on the testing results in Table 6.1, we can see that a strong argument exists: Ajax applications requiring high performance should utilize JSON rather than XML, because JSON improves the efficiency of an Ajax application by reducing network transfer time and client JavaScript processing time.

[48] http://www.w3.org/DOM/

6.1.3 Methodology

Transforming existing XML-based Ajax applications (such as the Ajax RSS Reader) to JSON-based Ajax applications is not straightforward. For an existing Ajax application (a common form of RIAs) changing the data format involves two procedures.

1. Converting the data from XML to JSON.
2. Changing the JavaScript code from the code which manipulates the XML data to the code which manipulates the JSON data.

However, programmers knowledgeable about XML may know nothing about JSON. In addition, programmers who implement the transformation may not be the original authors of the system; therefore, they require an understanding of all the interactions between the JavaScript code and the XML data. Such a process can clearly introduce defects; thus, we have designed a refactoring tool, XtoJ, to assist with the transformation of Ajax applications from utilizing XML to utilizing JSON.

We use refactoring to implement this transformation. Refactoring can be performed using either semi-automatic or fully-automatic approach (Mens & Tourwé, 2004). Our system is a semi-automated system as the transformation will often encompass functionality which cannot be automatically inferred by a program.

Our refactoring process contains three principle steps.

1. XML to JSON conversion. Converting an XML file into a JSON file is the first step in our refactoring process; the system is capable of achieving this conversion automatically. This transformation requires no "judgment calls" and is completely safe; hence, a completely automated approach is the best option.
2. JavaScript code transformation. After the XML file is converted into JSON, JavaScript code transformations can be completed automatically to transfer the JavaScript code which accesses the XML data to the functionally equivalent JavaScript code which processes the semantically-identical (to the XML) JSON data. This step is fully automated.
3. JavaScript code generation. The JavaScript code transformation only can convert the already existing JavaScript code which manipulates the existing XML data to manipulate the newly created JSON data. If more functionality is required, JavaScript code generation is invoked, in a semi-automated fashion, to produce the JavaScript code skeletons in accordance with users' requirements and users are asked to provide the "bodies" for these skeletons.

6.2 System Components

Our system has three components to perform the transformation; we now discuss each of them in detail.

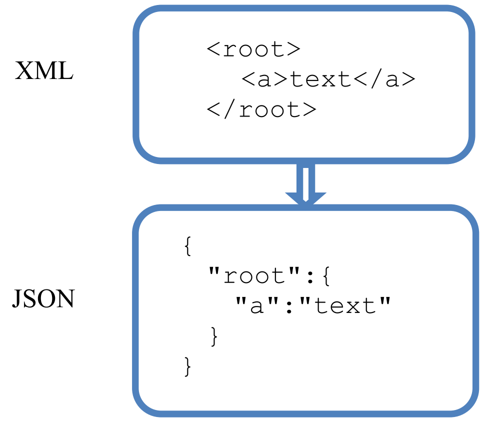
6.2.1 XML to JSON Converter

The XML to JSON Converter converts XML files into JSON files automatically. It is not necessary for programmers to know the syntax of XML or JSON, and the transformation rules between XML and JSON. We extended the grammar of JsonXml.js[49] which is a library to implement such transformation automatically.

Though XML and JSON have different syntax and structures, there are six basic rules for converting XML into JSON. For each rule, an XML version of the code and the corresponding JSON version of code are given, as well as the methods required to access the JSON version of the code, as an example. JavaScript provides an eval() function to convert a JSON string into an object. However, it is not secure since it can execute any piece of JavaScript code including malicious scripts. Hence, unless the client trusts the source of the data, it is advisable to use a JSON parser instead. To stop malicious scripts from executing, we use a JSON parser[50] to recognize and accept only valid JSON strings. Finally, grammatical statements covering the transformational rules are provided as general statements of the cases each transformational situation covers. The grammatical statements are Extended Backus-Naur Form (EBNF) (Attenborough, 2003) for XML 1.0 (W3C, 2008), namespace in XML 1.0 (W3C, 2009) and JSON[51]. To simplify the grammar, we only include directly utilized symbols and rules in the grammars.

6.2.1.1 Rules for Converting XML into JSON

Rule 1*:* An XML element that only has text



```
<root>
  <a>text</a>
</root>
```

```
{
  "root":{
    "a":"text"
  }
}
```

After the transformation, the following statements can be used to access the value of the node a.

```
var jsonObject = JSON.parse(xmlHttp.responseText);
var TextA = jsonObject.root.a;  //get "text"
```

49 http://michael.hinnerup.net/blog/2008/01/26/converting-json-to-xml-and-xml-to-json/
50 https://github.com/douglascrockford/JSON-js/blob/master/json2.js
51 http://www.json.org/

EBNF for XML 1.0

```
element: STag content ETag
STag: '<' Name  S? '>'
ETag: '</' Name S? '>'
content: (element | CharData)*
```

EBNF for JSON

```
object: '{' members? '}'
members: pair (',' pair)*
pair: string : value
value: string | object
string: '"' chars* '"'
```

Rule 2: An XML element that has text and attributes

XML

```
<root>
  <a name="value">text</a>
</root>
```

JSON

```
{
  "root": {
    "a":{
        "@name":"value",
        "#text":"text"
    }
  }
}
```

After the transformation, the following statements can be used to access the value and the attribute of the node a.

```
var jsonObject = JSON.parse(xmlHttp.responseText);
var TextA = jsonObject.root.a["#text"];    //get "text"
var AttributeA = jsonObject.root.a["@name"]; //get "value"
```

EBNF for XML 1.0

```
element: STag content ETag
STag: '<' Name (S Attribute)* S? '>'
Attribute: Name Eq AttValue
ETag: '</' Name S? '>'
content: (element | CharData)*
```

EBNF for JSON

```
object: '{' members? '}'
members: pair (',' pair)*
pair: ('"#text"'| string) : value
value: string | object
string: '"' ('@')? chars* '"'
```

Rule 3: An XML element that contains sub elements with the same names

XML

```
<root>
  <a>
     <b>text1</b>
     <b>text2</b>
  </a>
</root>
```

JSON

```
{
 "root": {
    "a": {
        "b":["text1","text2"]
       }
    }
}
```

After the transformation, the following statements can be used to access the value of the node b.

```
var jsonObject = JSON.parse(xmlHttp.responseText);
var Text1 = jsonObject.root.a.b[0]; //get "text1"
var Text2 = jsonObject.root.a.b[1]; //get "text2"
```

EBNF for XML 1.0

```
element: STag content ETag
STag: '<' Name S? '>'
ETag: '</' Name S? '>'
content: (element | CharData)*
```

EBNF for JSON

```
object: '{' members? '}'
members: pair (',' pair)*
pair: string : value
value: string | object | array
string: '"' chars* '"'
array: '[' elements? ']'
elements: value (',' value)*
```

Rule 4: An XML element that has contiguous text

XML

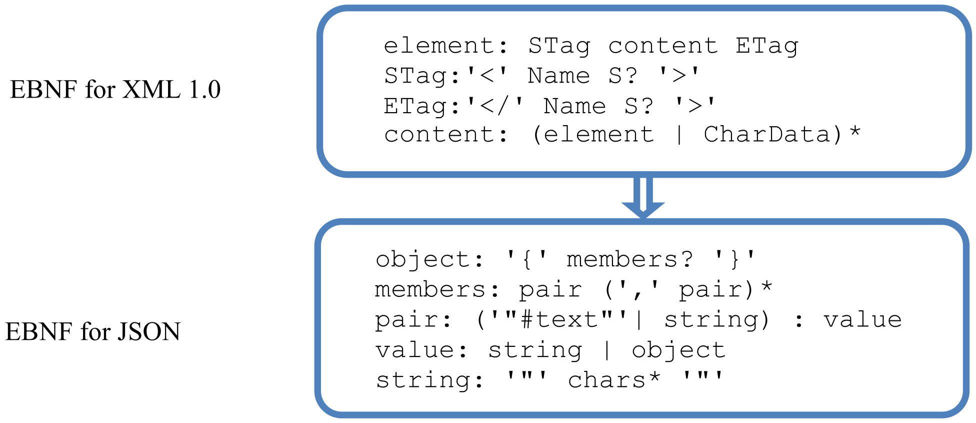
```
<root>
  <a>
     textA
     <b>textB</b>
  </a>
</root>
```

JSON

```
{
 "root": {
   "a": {
     "#text":"textA",
     "b":"textB",
    }
  }
}
```

After the transformation, the following statements can be used to access the value of the node a and the node b.

```
var jsonObject = JSON.parse(xmlHttp.responseText);
var TextA = jsonObject.root.a["#text"]; //get "textA"
var TextB = jsonObject.root.a.b;       //get "textB"
```

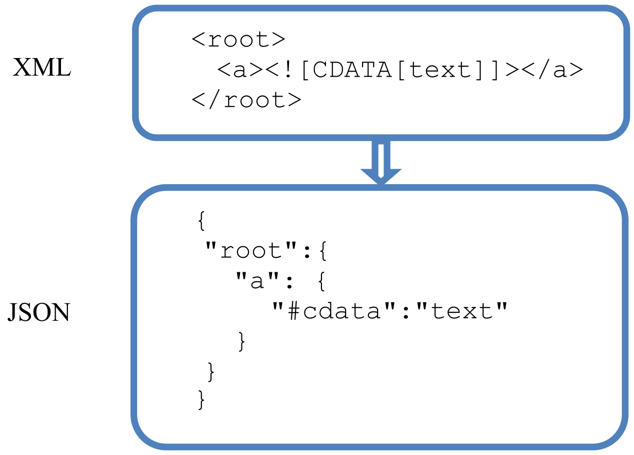

EBNF for XML 1.0

```
element: STag content ETag
STag:'<' Name S? '>'
ETag:'</' Name S? '>'
content: (element | CharData)*
```

EBNF for JSON

```
object: '{' members? '}'
members: pair (',' pair)*
pair: ('"#text"'| string) : value
value: string | object
string: '"' chars* '"'
```

Rule 5: An XML element that has a CDATA structure


XML

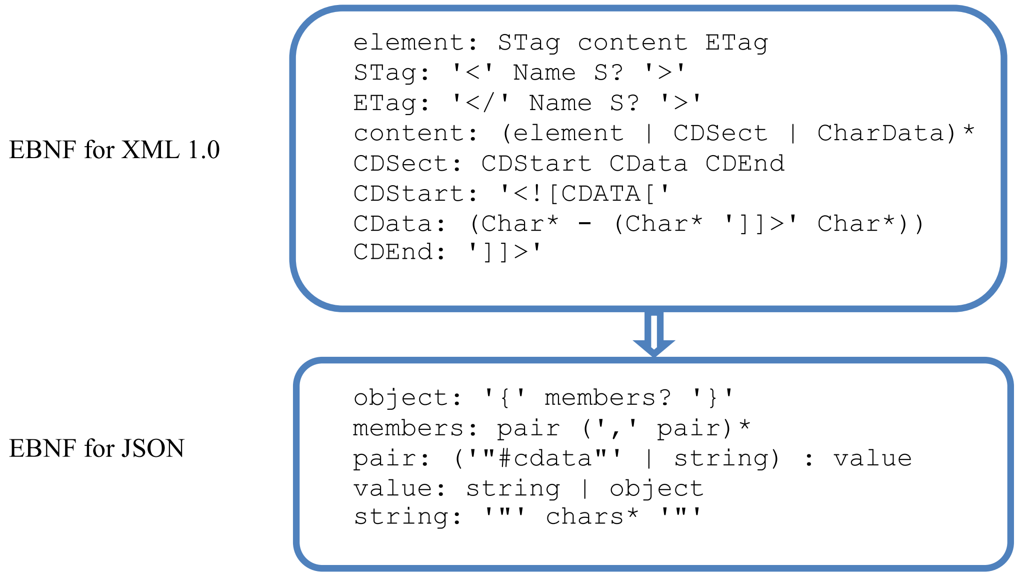
```
<root>
  <a><![CDATA[text]]></a>
</root>
```

JSON

```
{
 "root":{
   "a": {
      "#cdata":"text"
   }
 }
}
```

After the transformation, the following statements can be used to access the value of the node a.

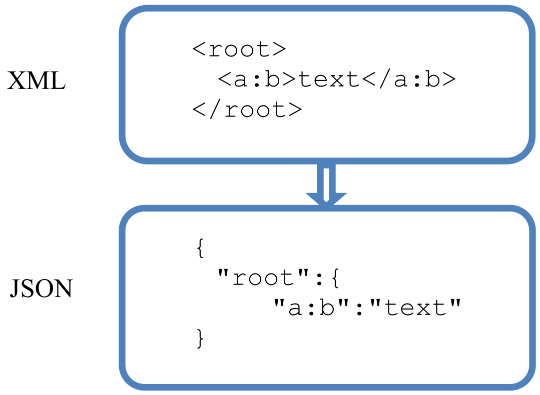
```
var jsonObject = JSON.parse(xmlHttp.responseText);
var TextA = jsonObject.root.a["#cdata"]; //get "text"
```



```
element: STag content ETag
STag: '<' Name S? '>'
ETag: '</' Name S? '>'
content: (element | CDSect | CharData)*
CDSect: CDStart CData CDEnd
CDStart: '<![CDATA['
CData: (Char* - (Char* ']]>' Char*))
CDEnd: ']]>'
```

```
object: '{' members? '}'
members: pair (',' pair)*
pair: ('"#cdata"' | string) : value
value: string | object
string: '"' chars* '"'
```

Rule 6: An XML element that uses namespaces



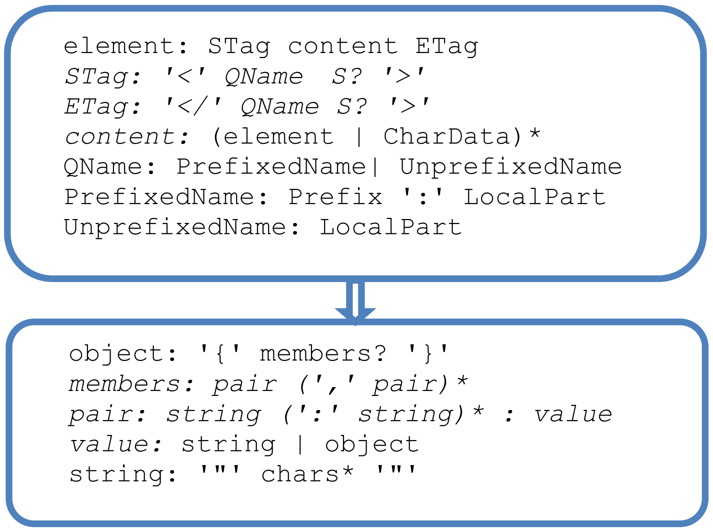
```
<root>
  <a:b>text</a:b>
</root>
```

```
{
  "root":{
      "a:b":"text"
}
```

After the transformation, the following statements can be used to access the value of the node a:b.

```
var jsonObject = JSON.parse(xmlHttp.responseText);
var TextAB= jsonObject.root["a:b"]; //get "text"
```

EBNF for namespaces in XML 1.0



```
element: STag content ETag
STag: '<' QName  S? '>'
ETag: '</' QName S? '>'
content: (element | CharData)*
QName: PrefixedName| UnprefixedName
PrefixedName: Prefix ':' LocalPart
UnprefixedName: LocalPart
```

EBNF for JSON

```
object: '{' members? '}'
members: pair (',' pair)*
pair: string (':' string)* : value
value: string | object
string: '"' chars* '"'
```

6.2.1.2 Example of System in Operation

We use the XML to JSON Converter to convert the RSS XML file in Section 6.1.1 to the RSS JSON file (rss.js). The structure of the created JSON file is as follows.

```
{
"channel":{
  "title":"http://www.imdb.com/rg/rss/news/news/ni9872099/fff",
  "link":"http://www.imdb.com/rg/rss/news-channel/news/",
  "description":"IMDb News",
  "language":"en",
  "copyright":"Copyright (C) 2011 IMDb.com, Inc.
    http://www.imdb.com/conditions",
  "image":{
    "title":"IMDb News",
    "url":"http://i.imdb.com/logo.gif",
    "link":"http://www.imdb.com/rg/rss/news-channel/news/"
  },
   "item":[{
     "title":"Actress McCormack Pregnant",
     "pubDate":"Fri, 22 Apr 2011 19:21:00 GMT",
     "link":"http://www.imdb.com/rg/rss/news/news/ni9872103/"
    },
    {
     "title":"Pattinson Made Up Royal Connection",
     "pubDate":"Fri, 22 Apr 2011 19:21:00 GMT",
     "link":"http://www.imdb.com/rg/rss/news/news/ni9872102/"
    },
   …
  ]}
}
```

6.2.2 JavaScript Code Transformer

The JavaScript Code Transformer automatically transforms JavaScript code used to access XML data into the JSON equivalent. It does this without requiring the programmers to have knowledge of how to access XML or JSON data. Despite changes to the data format, both the original code and the transformed version also require the same functionality.

To ensure the safety of the refactorings, the pre and post-conditions (Opdyke, 1992; Roberts, 1999) are established. The JavaScript Code Transformer is able to check the following pre and post-conditions.

Pre-conditions:

1. The syntax of the XML documents and the JavaScript code (DOM APIs) used to access the XML nodes is functionally correct.
2. The hierarchical depth of the XML documents is less than five. Based on our experience with XML documents in Ajax applications, this level of depth is sufficient to cover many existing XML documents. However, the tool can be easily modified to accommodate XML documents at greater depths if this is required.

Post-condition: The syntax of the converted JSON files and the JavaScript code used to access those JSON files is functionally correct.

ANTLR is used to implement the JavaScript Code Transformer. RhinoUnit[52], a framework for performing unit testing of JavaScript programs is used to ensure that the refactoring preserves the existing behavior of the program.

The JavaScript Code Transformer adopts a fully-automated approach to detect bad smells and rewrite the code.

1. Bad smell detection

The bad smells in our refactoring system arise from the inefficient set of XML DOM APIs used by JavaScript to manipulate XML nodes and attributes. XML DOM APIs includes: XML DOM properties (such as x.firstChild - x stands for any node), XML DOM methods (such as x.getElementsByTagName(“tagName”)) and XML attribute node methods (such as x.getAttribute(name)). The XML DOM APIs allow a variable to be used as a parameter for the methods to retrieve nodes and attributes. The bad smell detection phase can detect these statements to allow the code rewriting phase to successfully transform them into semantically-equivalent JSON statements.

The XML DOM APIs allow many different approaches to access nodes or attributes. For example, there are two approaches to access XML nodes, using the x.getElementsByTagName(tagName) method which returns all nodes with a specified tag name or using XML DOM properties (such as x.childNodes,

[52] http://code.google.com/p/rhinounit/

x.lastChild and x.nextSibling) to traverse the tree of the XML document. For example, the node "channel" in the rss.xml can be accessed by:

```
var node = RSSRequestObject.responseXML;
var channel = node.getElementsByTagName('channel').item(0);
```

or

```
var node = RSSRequestObject.responseXML;
var channel = node.documentElement.childNodes[0];
```

Additionally, three approaches can be used to access XML attributes, using the x.attributes property, x.getAttribute(name) and x.getAttributeNode(name) methods.

Since there is more than one approach to access XML nodes and attributes, programmers can use many different combinations of approaches to retrieve XML nodes or attributes they want to access. The bad smell detection phase can detect all of these different combinations.

2. Code rewriting

Unlike XML, JSON allows objects to contain other objects or arrays. Arrays can also contain other objects or arrays. JSON structure is accessed through dot or subscript operators from object to the member of the object to be retrieved. The name of the object or array is required to access its members. For example, the node "channel" in the rss.js can be accessed by:

```
var jsonObject = JSON.parse(RSSRequestObject.responseText);
var channel = jsonOjbect.channel;
```

Thus, the code rewriting phase transforms the bad smells - combination of XML DOM APIs that programmers use to access XML nodes or attributes - to the JavaScript statements used to access JSON structure. When a bad smell is detected, the JavaScript Code Transformer performs the following steps to rewrite code statements.

Step 1: Retrieve all the XML DOM APIs to determine the relationship between the nodes and the location of the XML node that is being accessed within the XML DOM.

Step 2: Based on the location determined in Step 1, the JavaScript Code Transformer traverses the XML document to retrieve the name of the node being accessed and the name of all the parent nodes of that node.

Step 3: Produce JavaScript statement(s) to access the node in JSON format based upon the information obtained in Step 2.

6.2.2.1 Grammar for the JavaScript Code Transformer

In this section, we provide the EBNF for the JavaScript code which accesses the XML and the JSON data respectively. The grammar for accessing XML nodes and XML attributes are different, thus, we show them separately. We extended the grammar from JavaScript.g[53] and again, to make the grammar simpler, we only include the directly utilized symbols and rules.

1. EBNF for JavaScript to access XML nodes

```
forStatement: 'for' '(' (forStatementInitialiserPart)? ';'
(forControl)? ';' (expression)? ')' statement;
forControl: Identifier '<' XMLHttpRequestName '.responseXML.'
('documentElement.')? ((('getElementsByTagName("' Identifier '")'
('[' (NumericLiteral | Identifier) ']' | 'item(' (NumericLiteral |
Identifier) ')') '.' )* ('getElementsByTagName("' Identifier '")'
| 'firstChild')) | ((('childNodes' ('[' (NumericLiteral |
Identifier) ']' | 'item(' (NumericLiteral | Identifier) ')')) '.'
)* ('childNodes' | 'firstChild'))) '.' 'length';
variableStatement: 'var' variableDeclarationList ';';
variableDeclarationList: variableDeclaration (','
variableDeclaration)*;
variableDeclaration: Identifier initialiser?;
Initialiser: '=' XMLhttprequestName '.responseXML.'
('documentElement.')? ('getElementsByTagName("' Identifier '")'
('[' (NumericLiteral | Identifier) ']' | 'item(' (NumericLiteral |
Identifier) ')') ('.')? )* ('childNodes' ('[' (NumericLiteral |
Identifier) ']' | 'item(' (NumericLiteral | Identifier) ')')
('.')? )* ('firstChild' ('.')?) ('nodeValue' | 'data');
```

2. EBNF for JavaScript to access XML attributes. (All the grammar rules for accessing XML attributes are the same as accessing XML nodes except the rule “Initialiser”, shown as follows.)

```
Initialiser: '=' XMLhttprequestName '.responseXML.'
('documentElement.' )? ('getElementsByTagName("' Identifier '")'
('[' (NumericLiteral | Identifier) ']' | 'item(' (NumericLiteral |
Identifier) ')') ('.')? )* ('childNodes' ('[' (NumericLiteral |
Identifier) ']' | 'item(' (NumericLiteral | Identifier) ')')
('.')? )* ('firstChild' ('.')?) ((('getAttributeNode("' Identifier
'")') |(attributes('[' (NumericLiteral | Identifier) ']' | 'item('
(NumericLiteral | Identifier) ')')) | ('attributes' '.'
'getNamedItem("' Identifier '")')) '.' nodeValue) | (getAttribute
'("' Identifier '")');
```

3. EBNF for JavaScript to access JSON

```
forStatement: 'for' '(' (forStatementInitialiserPart)? ';'
(forControl)? ';' (expression)? ')' statement;
forControl: Identifier '<' XMLHttpRequestName '.' responseText
'.' (Identifier '.')+ length;
variableStatement: 'var' variableDeclarationList ';';
variableDeclarationList: variableDeclaration (','
variableDeclaration)*;
variableDeclaration: Identifier initialiser?;
Initialiser: '=' 'JSON.parse(' XMLHttpRequestName '.responseText)'
(('.' Identifier) | ('.' Identifier '[' Identifier | ('"' #Text
'"') | ('"@' Identifier '"') | ('"' #cdata '"') | ('"' Identifier
'":"' Identifier '"') ']'))*;
```

[53] http://www.antlr.org/grammar/1206736738015/JavaScript.g

6.2.2.2 Example of System in Operation

Again, we use the example in Section 6.1.1 to illustrate how the JavaScript Code Transformer works. After an XML file is successfully converted into a JSON file by the XML to JSON Converter, the JavaScript Code Transformer takes an HTML or JavaScript file that retrieves the data from an XML file as input and outputs an HTML or JavaScript file that retrieves data from the converted JSON file. The process of transformation is as follows.

Source code preparation: The transformation is on JavaScript code, thus, if the input file is HTML, all the HTML elements are removed. However, the HTML elements embedded in the JavaScript snippet will stay the same. In the example, the input file is an HTML file.

Copy propagation transformation: This transformation "eliminates cases in which values are copied from one location or variable to another" (Hagen, 2006). To prepare for the transformation, statements that are used to retrieve the node properties are combined with statements that are used to access the value of the node. For example, the variable "node" and "channel" are eliminated in the following statements.

```
var node = RSSRequestObject.responseXML;
var channel = node.getElementsByTagName('channel').item(0);
var title = channel.getElementsByTagName('title').item(0).
            firstChild.data;
```

are changed to:

```
var title = RSSRequestObject.responseXML.
            getElementsByTagName('channel').item(0).
            getElementsByTagName('title').item(0).firstChild.data;
```

JavaScript code transformation: The JavaScript parser generated by ANTLR is used to parse the JavaScript code for accessing the XML file. If a bad smell (the combination of XML DOM APIs that used to access XML nodes or attributes) is found. The system rewrites the code according to the transformation rules. Figure 6.3 shows the JavaScript code to process the JSON file after refactoring.

The JavaScript code transformation is comprise of three steps.
Step 1: Change responseXML into responseText. In the JavaScript code for JSON (Figure 6.3), the responseXML property of the XMLHttpRequest object (RSSRequestObject) is changed to the responseText property, which gets the response (non-XML) and returns a string (JSON).

Step 2: Change the JSON string into an object. The JSON parser[54] is used to change the JSON string into an object (jsonObject).

Step 3: Change code statements used to access the values. JavaScript code that accesses the value of the nodes <title>, <link> and <item> (including child nodes

[54] https://github.com/douglascrockford/JSON-js/blob/master/json2.js

<title>, <link> and <pubDate>) in the RSS XML file are transformed to access the corresponding members of the objects in the RSS JSON file for this example. The <item> node is converted to an object containing three members “title”, “link” and “pubDate”. If the “item” object is an array, an alternative mechanism is required to access its members. Thus, the value of typeof(json.channel.item[0]) is used to check whether the “item” object is an array. An “undefined” value indicates that the “item” object is not an array; its members are accessed without iteration. A numbered value indicates that “item” object is an array; iteration is required to access the members.

Cleanup: If the input file is HTML, the HTML elements are added back and the output file is generated.

```
var jsonObject = JSON.parse(RSSRequestObject.responseText);
var title = json.channel.title;
var link = json.channel.link;

content = '<div class="channeltitle"><a href="'+link+'">'+title+'</a></div><ul>';

if(typeof(json.channel.item[0]) == 'undefined'){
   var itemTitle = json.channel.item.title;
   var itemLink = json.channel.item.link;
   try{
      var itemPubDate = '<font color=gray>['+json.channel.item.pubDate+']';
   }catch (e){
       var itemPubDate = '';
   }
   content += '<li>'+itemPubDate+'</font><a href="'+itemLink+'">'+itemTitle+'</a></li>';
   }
else{
   for(var n=0; n < json.channel.item.length; n++){
      var itemTitle = json.channel.item[n].title;
      var itemLink = json.channel.item[n].link;
      try{
          var itemPubDate = '<font color=gray>['+json.channel.item[n].pubDate+'] ';
      }catch (e){
          var itemPubDate = '';
      }
      content += '<li>'+itemPubDate+'</font><a href="'+itemLink+'">'+itemTitle+'</a></li>';
   }
}
```

Figure 6.3 The JavaScript Code for Accessing the JSON File after the Transformation

6.2.3 JavaScript Code Generator

The JavaScript Code Generator is an optional component in our transformation system. After refactoring the existing application, the JavaScript Code Generator can be used to generate JavaScript code skeletons to access a JSON file when more functionality is required. Unlike the JavaScript Code Transformer which is fully-automated, the JavaScript Code Generator is semi-automated as the programmer must supply information to allow the code to be constructed safely. Specifically, the user must:

1. select an XML file to be converted to a JSON file;
2. select groups of data (nodes or attributes) required to be accessed;
3. provide an explanation of the conditions under which the data can be safely accessed; and
4. select the format for the output to

- print the data,
- store the data into an array, and
- display the data through a form.

Given this information, the system creates the JavaScript code for processing the JSON structure produced in Step 1. The generated code is considered a skeleton of the final program as the system has no way of understanding the domain of the application. The final (manual) step is for the programmer to add any domain specific component.

For each node selected by the programmer, the JavaScript Code Generator performs the following steps to generate the JavaScript code for accessing the converted JSON file.
Step 1: Retrieve the node selected by the programmer and determine the location of the XML node within the XML DOM.

Step 2: Using the location determined in Step 1, the JavaScript Code Generator traverses the XML document to retrieve the name of the node being selected and the name of all the parent nodes of that node.

Step 3: Produce JavaScript statement(s) to access the node in JSON format based upon the information obtained in Step 2.

6.2.3.1 Patterns for Generating JavaScript Code

The JavaScript Code Generator creates code for:
1. accessing node values or attributes for a single node; and
2. traversing nodes to get values or attributes for multiple nodes.

The following section discusses different patterns to generate JavaScript code for accessing a JSON file. As in previous sections, we provide the EBNF for XML 1.0 (W3C, 2008) and the EBNF for the generated JavaScript code for each pattern. We only include the directly utilized symbols and rules of the grammar. The following explicit definitions are required with respect to EBNF for the generated JavaScript Code.

JsonObject: the object name after converting a JSON string to an object.
nodeMembers: the name(s) of the node(s) in an XML file.
attributeMember: the attribute name(s) in an XML file.
conditions: the condition(s) under which it is safe to access the data.

1. EBNF for XML 1.0

```
document: prolog element Misc*
element: EmptyElemTag | STag content ETag
STag:'<' Name (S Attribute)* S? '>'
Attribute: Name Eq AttValue
ETag:'</' Name S? '>'
content: (element | CharData | Reference | CDSect | PI | Comment)*
```

2. EBNF for generated JavaScript code to access a single node

- Accessing the value of a node

```
output: JsonObject nodeMembers+;
JsonObject: Identifier;
nodeMembers: ('.' Identifier) | ('.' Identifier '[' Identifier |
('"' #Text '"') | ('"@' Identifier '"') | ('"' #cdata '"') | ('"'
Identifier '":"' Identifier '"') ']');
```

- Accessing the attribute of a node

```
output: JsonObject nodeMembers+ '.' attributeMember
JsonObject: Identifier;
nodeMembers: ('.' Identifier) | ('.' Identifier '[' Identifier |
('"' #Text '"') | ('"@' Identifier '"') | ('"' #cdata '"') | ('"'
Identifier '":"' Identifier '"') ']');
attributeMember: Identifier;
```

3. EBNF for generated JavaScript code to traverse nodes

- Accessing the value of nodes

```
output: forStatement | ifStatement;
forStatement: 'for(' forInStatementInitialiserPart  ';' forControl
';' expression ')' statement;
statement: forStatement | ifStatement | accessStatement;
ifStatement: 'if(' ifExpression ')' statement 'else' statement;
forControl:  Identifier  '<' JsonObject nodeMembers+ '.length'
ifExpression:   ('typeof('   JsonObject   nodeMembers+   '[0]   ==
undefined)') | conditions
conditions: expression;
accessStatement: 'document.write(' JsonObject nodeMembers+ ')'
JsonObject: Identifier
nodeMembers:  ('.' Identifier) |('.' Identifier '[' Identifier |
('"' #Text '"') | ('"@' Identifier '"') | ('"' #cdata '"') | ('"'
Identifier '":"' Identifier '"') ']');
```

- Accessing the attributes of nodes

```
output: forStatement | ifStatement;
forStatement: 'for(' forInStatementInitialiserPart  ';' forControl
';' expression ')' statement;
statement: forStatement | ifStatement | accessStatement;
ifStatement: 'if(' ifExpression ')' statement 'else' statement
forControl:  Identifier  '<' JsonObject nodeMembers+ '.length';
ifExpression: ('typeof(' JsonObject nodeMembers+ '[0] ==
undefined)') | conditions;
accessStatement:  'document.write('  JsonObject  nodeMembers+  '.'
attributeMember) ')';
conditions: expression;
JsonObject: Identifier;
nodeMembers:  ('.' Identifier) |('.' Identifier '[' Identifier |
('"' #Text '"') | ('"@' Identifier '"') | ('"' #cdata '"') | ('"'
Identifier '":"' Identifier '"') ']');
attributeMember: Identifier;
```

6.2.3.2 Example of System in Operation

In this section, we reuse the RSS XML file from IMDb[55] to illustrate the operation of the JavaScript Code Generator. There are six steps (five input steps plus the code production step) for programmers to produce JavaScript code as shown in Figure 6.4 from scratch.

Step 1: Select the XML file to be converted. The RSS XML file is converted into the RSS JSON file (rss.js) by the XML to JSON Converter.

Step 2: Select the groups of data (node values or attributes) to be accessed. In this example, we have selected to access the values of the nodes <title> and <link> whose parent node is <channel> and all the values of the nodes whose parent node is <item>.

Step 3: The programmer provides the conditions for accessing the data (for example, only the feeds published in the morning are retrieved). This is optional; hence we have omitted it here for the sake of brevity.

Step 4: Select the format for output. We have chosen to print the retrieved data (using document.write).

Step 5: The system, using these inputs (Steps 1-4), automatically generates JavaScript code for accessing the new JSON object, as shown in Figure 6.4.

To explain Step 5 further, we provide a brief overview of the generated code.

Line1-7: Creates an XMLHttpRequest object for different browsers.

- If the web browsers are Internet Explorer 5 or 6, an XMLHttpRequest object is created using ActiveX controls.
- If the web browsers are Internet Explorer 7, 8 or 9, Mozilla Firefox, Google Chrome, Opera and Safari, an XMLHttpRequest object is created using a native object.

Line 8 and 28: Send a request to the server. Line 18 makes a GET request for the URL: "rss.js" and line 28 sends the request to the server using the send() function.

Line 9 and 10: Handle properties of the XMLHttpRequest. Ultimately these lines indicate when the response is completed and all the data has been received.

Line 11: Checks the HTTP status. The request is completed “correctly” when the value is 200.

Line 13: Converts the JSON string (rss.js) to a JSON object named “jsonObject” using the JSON parser[56].

[55] http://www.imdb.com/

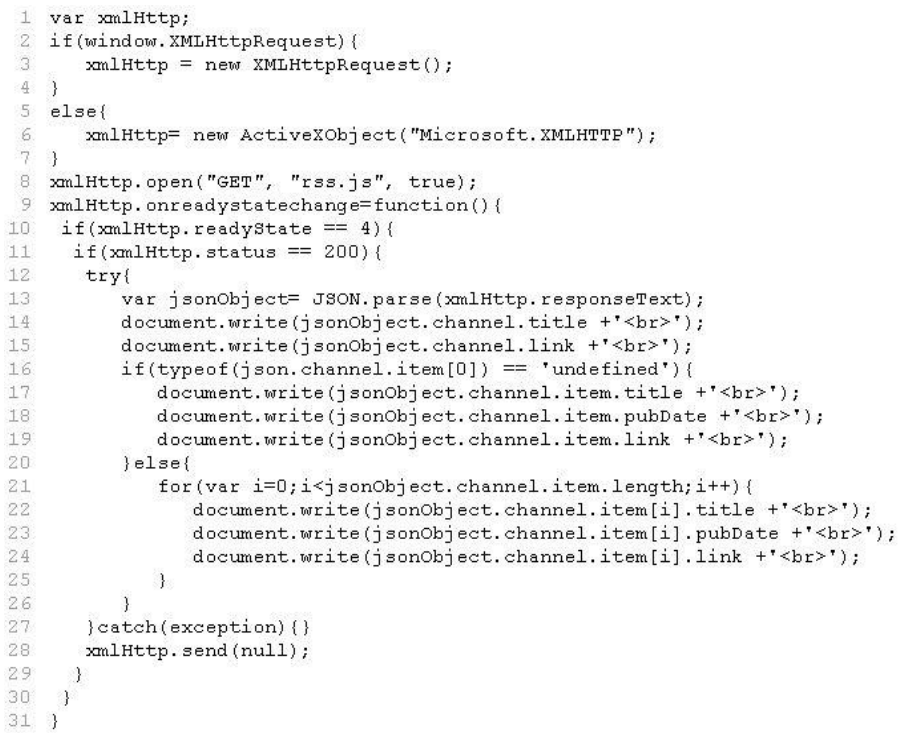
[56] https://github.com/douglascrockford/JSON-js/blob/master/json2.js

Line 14-15: Retrieve the members of “title” and “link” in the object “channel” (as stated Step 2). All the JSON objects are retrieved by the names of their objects. These objects are actually the nodes’ names or attributes’ names in the RSS XML file. After retrieval, these objects are outputted via document.write (as stated in Step 4).

Line 16: Checks whether the “item” object is an array.

Line 17-19: If the “item” object is not an array, the members of “title”, “link” and “pubDate” are retrieved.

Line 21-25: If the “item” object is an array, the code traverses the “item” object and accesses the members of “title”, “link” and “pubDate”, for every object in the “item” object. After retrieval, these objects are outputted via document.write (as stated in Step 4).



```
 1  var xmlHttp;
 2  if(window.XMLHttpRequest){
 3     xmlHttp = new XMLHttpRequest();
 4  }
 5  else{
 6     xmlHttp= new ActiveXObject("Microsoft.XMLHTTP");
 7  }
 8  xmlHttp.open("GET", "rss.js", true);
 9  xmlHttp.onreadystatechange=function(){
10   if(xmlHttp.readyState == 4){
11    if(xmlHttp.status == 200){
12     try{
13        var jsonObject= JSON.parse(xmlHttp.responseText);
14        document.write(jsonObject.channel.title +'<br>');
15        document.write(jsonObject.channel.link +'<br>');
16        if(typeof(json.channel.item[0]) == 'undefined'){
17           document.write(jsonObject.channel.item.title +'<br>');
18           document.write(jsonObject.channel.item.pubDate +'<br>');
19           document.write(jsonObject.channel.item.link +'<br>');
20        }else{
21           for(var i=0;i<jsonObject.channel.item.length;i++){
22              document.write(jsonObject.channel.item[i].title +'<br>');
23              document.write(jsonObject.channel.item[i].pubDate +'<br>');
24              document.write(jsonObject.channel.item[i].link +'<br>');
25           }
26        }
27     }catch(exception){}
28     xmlHttp.send(null);
29    }
30   }
31  }
```

Figure 6.4 The Generated JavaScript Code for Accessing the RSS JSON File

Step 6: Add domain specific components. In the JavaScript code presented in Figure 6.3, the variable “content” is created and used to output HTML elements, which have the values of the retrieved objects from the RSS JSON file (rss.js). To produce the final program, the programmer modifies the generated code (Figure 6.4) according to the specific requirements. Thus, instead of outputting the different members of the “channel” object and the “item” object, a variable “content” is used to provide the output.

6.3 Evaluation

To evaluate our technique, we randomly select ten XML-based Ajax applications from the Internet to test the effectiveness of the refactoring system. Because no new functionality is added to the existing ten applications, the JavaScript Code Transformer is used to automatically change the ten applications from XML-based to JSON-based. Subsequently, we test the "response time" of each Ajax application utilizing XML and utilizing JSON.

6.3.1 Methodology

We use the Client/Server architecture to send XML or JSON data from a server to a client. The process of transferring data is as follows.

1. After a TCP connection is created, the client creates an XMLHttpRequest object and sends an HTTP request by using the XMLHttpRequest object to the server. In addition, the client specifies what type of data is to be retrieved (XML or JSON).
2. The server processes the request and creates an HTTP response message. The responseXML and responseText properties of the XMLHttpRequest object are used to retrieve the requested XML or JSON data from the HTTP response on the client side.

We measure the "response time" as an amalgamation of the network transfer time, the network latency time, the server response time and the client processing time. It starts from the time that the XMLHttpRequest object is created on the client side to the time that all the data has been retrieved from the server and processed by the client.

Tests are executed in the following environment.

- Server: Intel(R) Core (TM) 2 Quad CPU Q6600 @2.4GHz, with 4 GB of RAM running Microsoft Windows XP Professional, Service Pack 3.
- Client: Intel(R) Core (TM) 2 CPU 6300 @1.86GHz, with 2 GB of RAM running Microsoft Windows 7 Professional.

In addition, we use two different browsers: Firefox 4.0 and Internet Explorer 8.0 to ensure that no browser-specific bias is introduced. Finally, each test is executed 100 times to ensure that any extraneous timing issues are minimized.

6.3.2 Testing Results for Ten Ajax Applications

To determine the improvement in performance, we test with different numbers of objects being transferred. Although Ajax is used to perform partial updates of the user interface, a partial update does not mean that only a small amount of data is retrieved. In addition, a partial update does not imply that only a small section of the application is updated. For example, similar to Microsoft Excel, online spreadsheet applications such as EditGrid[57], Google Spreadsheets[58] and Zoho

[57] http://www.editgrid.com/
[58] http://www.google.com/

Sheet[59] require most of the user's browser window to be updated with hundreds of cells (objects or properties) when the user switches between workbooks. Online mapping applications such as Google Maps[60] and Yahoo Maps[61] also require that the majority of the user's browser window be updated each time the user pans, zooms, or performs any other activity. Other Ajax-enabled websites such as Facebook[62] (switching between most pages is done using Ajax); Dell's Online Store[63] (customization of the entire computer system is done through Ajax-enabled technology); and Gmail[64] (most navigation within Gmail is via Ajax) require hundreds of objects or properties be loaded as quickly as possible to provide users with an uninterrupted user experience similar to that achieved in desktop applications. As more companies start transforming existing traditional web applications into Ajax-enabled applications and as Ajax applications grow in complexity, the volume of Ajax applications that will request a large number of objects will also grow.

As stated above, many large Ajax-based applications regularly transfer a large number of objects, hence we elect to start our investigation from 100 objects (lower end); and provide values up to 1000 (upper end), which perhaps represents the maximum volume of transfers that are likely to be witnessed from the current generation of Ajax-based applications. Table 6.2 lists the size information of the XML documents used in our trials.

Table 6.2 The XML Documents Size for the Ten Tested Applications

	App	100 Objects	500 Objects	1000 Objects
1	w3schools.com[65]	19KB	94 KB	187 KB
2	ibm.com[66]	7 KB	34 KB	68 KB
3	developer.com[67]	79 KB	391 KB	781 KB
4	captain.at[68]	8 KB	39 KB	78 KB
5	JavaScriptkit.com[69]	117 KB	583 KB	1165 KB
6	Understanding AJAX: Using JavaScript to Create Rich Internet Applications (Eichorn, 2007)	6 KB	27 KB	53 KB
7	ibm.com[51]	9 KB	43 KB	86 KB
8	sitepoint.com[70]	7 KB	32 KB	63 KB
9	xml.com[71]	8 KB	40 KB	79 KB
10	Brainjar.com[72]	56 KB	277 KB	554 KB

59 http://www.zoho.com/
60 http://maps.google.ca/
61 http://ca.maps.yahoo.com/
62 http://www.facebook.com/
63 http://www.dell.com/
64 https://mail.google.com/mail/help/intl/en/about.html
65 http://www.w3schools.com
66 https://www.ibm.com/developerworks/xml/
67 http://www.developer.com/
68 http://www.captain.at/
69 http://www.javascriptkit.com/
70 http://sitepoint.com/
71 http://www.xml.com/
72 http://www.brainjar.com/

The size of these documents is in many ways arbitrary, as we know of no reliable information on the (average) size of XML documents in web applications. However, a quick search of the Internet can find significant numbers of applications using XML documents which are several orders of magnitude larger than the sizes used in our trials. Table 6.3 provides some examples of web applications that utilize "large" XML documents (Ng et al., 2006).

Table 6.3 The Size of Some Large XML Documents

Data sets	Size
XMark (Schmidt et al., 2002)	97 MB
DBLP[73]	42 MB
Shakespeare[74]	7.8 MB
SwissProt[75]	21 MB
TPC-H[76]	34 MB
Weblog[77]	30 MB

For brevity purposes, Table 6.4 shows the response time (ms) for retrieving the XML and the JSON data from the server for three different quantities of objects (100, 500 and 1000 objects). Again, this task is repeated 100 times, and the average is used in all calculations. This mechanism provides a robust estimation of the response time as a number of factors exist which impact the individual results but are outside the control of the experimenters. The RMD of the response time of accessing XML data and JSON data is utilized.

Table 6.4 Testing Results for the Ten Ajax Applications

	Number of Objects	Firefox 4.0			IE 8.0		
		XML (ms)	JSON (ms)	RMD	XML (ms)	JSON (ms)	RMD
1	100	5.04	1.26	1.20	5.63	1.54	1.14
	500	20.59	3.20	1.46	24.51	4.42	1.39
	1000	48.09	5.40	1.60	47.93	8.28	1.41
2	100	5.83	2.23	0.89	4.57	2.73	0.50
	500	24.81	8.11	1.01	19.50	12.54	0.43
	1000	47.47	15.95	0.99	38.21	25.02	0.42
3	100	10.03	4.67	0.73	13.28	6.72	0.66
	500	45.34	19.08	0.82	70.16	35.31	0.66
	1000	89.32	37.5	0.82	147.65	72.5	0.68
4	100	3.54	1.38	0.88	8.89	1.60	1.39
	500	12.76	3.58	1.12	40.89	5.06	1.56
	1000	24.26	5.96	1.21	80.83	9.67	1.57
5	100	20.94	6.21	1.09	31.23	8.13	1.17
	500	117.90	28.02	1.23	166.60	44.82	1.15
	1000	215.76	64.10	1.08	375.20	93.72	1.20

[73] http://dblp.uni-trier.de/

[74] http://www.cs.wisc.edu/niagara/data/shakes/shakspre.htm

[75] http://www.expasy.ch/sprot/

[76] http://www.tpc.org/tpch/default.asp

[77] http://httpd.apache.org/docs/logs.html

6	100	3.03	1.21	0.86	3.22	1.48	0.74
	500	10.82	2.93	1.15	12.37	4.46	0.94
	1000	19.74	4.62	1.24	23.65	8.11	0.98
7	100	2.33	1.37	0.52	2.93	1.77	0.49
	500	7.07	3.75	0.61	10.93	5.95	0.59
	1000	12.73	7.02	0.58	20.81	11.71	0.56
8	100	3.04	1.35	0.77	5.78	2.81	0.69
	500	11.24	4.07	0.94	21.40	9.53	0.77
	1000	21.66	6.94	1.03	40.79	17.66	0.79
9	100	5.32	0.85	1.45	10.73	1.45	1.52
	500	22.62	2.92	1.54	54.45	4.09	1.72
	1000	53.25	4.77	1.67	119.50	7.78	1.76
10	100	22.92	3.47	1.47	42.61	6.68	1.46
	500	118.60	19.20	1.44	216.40	32.90	1.47
	1000	232.00	34.10	1.49	441.10	65.80	1.48

From this table, we can see that the JSON version of every program is consistently faster than the XML version; these results broadly correspond to the results provided by Nurseitov et al. (2009). Since we are able to "reproduce" the efficiency saving from the work of Nurseitov et al. (2009), we believe that this illustrates that our transformation process is successful in refactoring Ajax code for efficiency. These results show the browser version does have an impact on the response time. Firefox 4.0 generally outperforms IE 8 in both XML and JSON versions thanks to a more modern JavaScript engine. However, both browser versions benefit when switched from XML to JSON.

6.3.3 Variables Influencing the Response Time

During the testing of the ten Ajax applications, we find three additional variables that significantly affect the response time for accessing XML and JSON data and thus affecting the efficiency improvement rate of our transformation system. We take the Ajax RSS Reader in Section 6.1.1 as an example to demonstrate their impact.

1. The number of objects

To test the impact of the number of objects (or nodes), we investigate different number of feeds in an RSS file for popular websites. The number of feeds in an RSS file varies from website to website; however, a quick search of the Internet can find many websites using 50 or more feeds in their RSS files. Some websites have fixed number of feeds, such as IMDb[78] (50 feeds), the New York Times-books[79] (50 feeds), investopedia[80] (60 feeds), TUAW[81] (40 feeds) and Engadget[82] (40 feeds). Some websites change the number of feeds every day. For example Gizmodo[83] may have up to 100 feeds according to our observations. We test the XML and JSON version of the Ajax RSS Reader code by retrieving 50,

[78] http://www.imdb.com/
[79] http://www.nytimes.com/pages/books/index.html
[80] http://www.investopedia.com/
[81] http://www.tuaw.com/
[82] http://www.engadget.com/
[83] http://ca.gizmodo.com/

100, 150 and 200 feeds (objects) from the RSS XML file of IMDb and the converted RSS JSON file. For each trial, we measure the response time for the XML version and JSON version of the program, using the methodology described in Section 6.3.2. For the XML version, the response time for accessing different numbers of child nodes (<title>, <title> + <link>, and <title> + <link> + <pubDate>) within the node <item> is measured. For the JSON version, the response time for accessing the same numbers of objects as the XML version of the code is measured. Table 6.5 indicates the influence of the different number of objects tested in Firefox 4.0. As more objects are accessed, the response time for both the XML and JSON version of the code increases.

Table 6.5 Testing Results for Different Number of Objects

Number of the Objects (feeds)	Objects Accessed within the Object "item"	XML (ms)	JSON (ms)	RMD
50	title	2.68	1.27	0.71
	title, link	3.01	1.30	0.79
	title, link, pubDate	3.29	1.34	0.84
100	title	4.94	2.15	0.79
	title, link	5.48	2.25	0.84
	title, link, pubDate	6.26	2.41	0.89
150	title	6.59	2.46	0.91
	title, link	7.60	2.74	0.94
	title, link, pubDate	8.50	2.95	0.97
200	title	8.28	2.74	1.01
	title, link	9.35	2.92	1.05
	title, link, pubDate	11.02	3.38	1.06

2. The structure of the XML and JSON data

XML documents have a hierarchical structure; accessing different nodes in different hierarchical depths affects the response time. We test the execution time for accessing the text of the node <title> whose parent node is <channel> and the text of the node <title> whose parent node is <image>. These nodes have different depths.

The JSON data is accessed by the dot operator. Accessing an object and accessing the objects within that object results in different response time. We tested the execution time for the JSON version of the code to access the jsonObject.channel.title object and the jsonObject.channel.image.title object. Each trial is run 10,000 times in Firefox 4.0 to produce a stable mean value. Table 6.6 clearly shows that for the XML version of the code, accessing nodes in deeper structures increases the response time of the browsers; for the JSON version, accessing an object directly is faster than accessing any objects inside the object.

In addition, accessing the attribute of a node and accessing the text of a node in XML files also leads to different response time. To test the execution time for accessing the attribute of a node, we manually add an attribute "language" to the node <title> whose parent node is <channel>. Subsequently, we test the execution time for accessing the attribute "language" of the node <title> in the XML file. The object "language" within the object "title" in the JSON file is also tested.

Again, each trial is run 10,000 times in Firefox 4.0 to produce a stable mean value. As can be seen from Table 6.6, accessing the attribute of a node is faster than accessing the text of a node for the XML version; however, for the JSON version, accessing the converted attribute object within an object takes a similar time (in absolute terms) as accessing that object.

Table 6.6 Testing Results for Accessing Different Objects in Different Structures

Object	Parent Object	Depth	XML (ms)	JSON (ms)	RMD
title	channel	1	1.90	0.06	1.88
title	image	2	2.94	0.08	1.89
attribute	channel	1	1.77	0.06	1.87

3. The length of the text in a node

To analyze the influence of the length of the text in a node, we test the response time of accessing the text of the node <title> in the XML file and the "title" object in the JSON file by manually changing the length of the text. We use the same methodology as discussed in Section 6.3.2 and the results are shown in

Table 6.7. As the text length increases, the longer it takes for the browser to respond to both the XML and the JSON version of code.

Table 6.7 Testing Results for Accessing a Node with Different Lengths

Length of the Text	XML (ms)	JSON(ms)	RMD
50 Chars	2.11	1.15	0.59
1000 Chars	2.87	1.27	0.77

As can be seen from all the results, a number of factors, beyond the number of object retrieved, significantly impact the efficiency of RIAs using either XML or JSON. Given almost any combination of these factors, implies that significant performance differences will exist between semantically equivalent implementations based upon either data format. However, in every situation, utilizing JSON is more efficient than utilizing XML. Hence, these figures provide an empirical proof that a large number of situations exist where it is worthwhile, from a performance viewpoint, to transform an existing application from an XML-based application to a JSON-based application.

Chapter 7 Refactoring Traditional Forms into Ajax-enabled Forms

The Web still has many traditional forms, and transforming these forms into Ajax-enabled forms is not straightforward. HTML forms are common for interactive web applications; it has different HTML elements for the user on the browser (client side) to fill out. HTML forms can contain input elements (including different types: text field, password field, button, radio button, check box, reset, submit, image, file and hidden field), text areas and select menus. Form submission can be invoked using an input element, a button element, a label element, text or an image. To transform traditional HTML forms into Ajax-enabled forms, programmers would need to know how to manipulate all these input elements using JavaScript or a JavaScript Ajax-library and to add the appropriate Ajax-enabled statements in the right locations.

Hence, to aid with this transformation process, refactoring can be used. In this Chapter, we extend the refactoring idea to allow programmers to refactor traditional web forms into Ajax-enabled forms using a semi-automated refactoring tool, FTT. The purpose of our refactoring is to improve the efficiency of web forms.

The remainder of this chapter is as follows: Section 7.1 and Section 7.2 provide an example of a traditional web form and present our refactoring approach. Section 7.3 describes the three components of our refactoring system. Section 7.4 shows the results of transforming two example forms into Ajax-based forms.

7.1 Motivating Example

In this section, we utilize JspCart[84] to demonstrate efficiency problems with forms in traditional web applications. JspCart is an open source shopping cart application. It is developed using JSP (JavaServer Pages) (Bergsten, 2000) and JavaBeans (Englander, 1997), and runs on Apache Tomcat[85] and MySQL[86]. Figure 7.1 shows the user registration web page of JspCart (Signup.jsp).

7.1.1 Problem 1: Submission

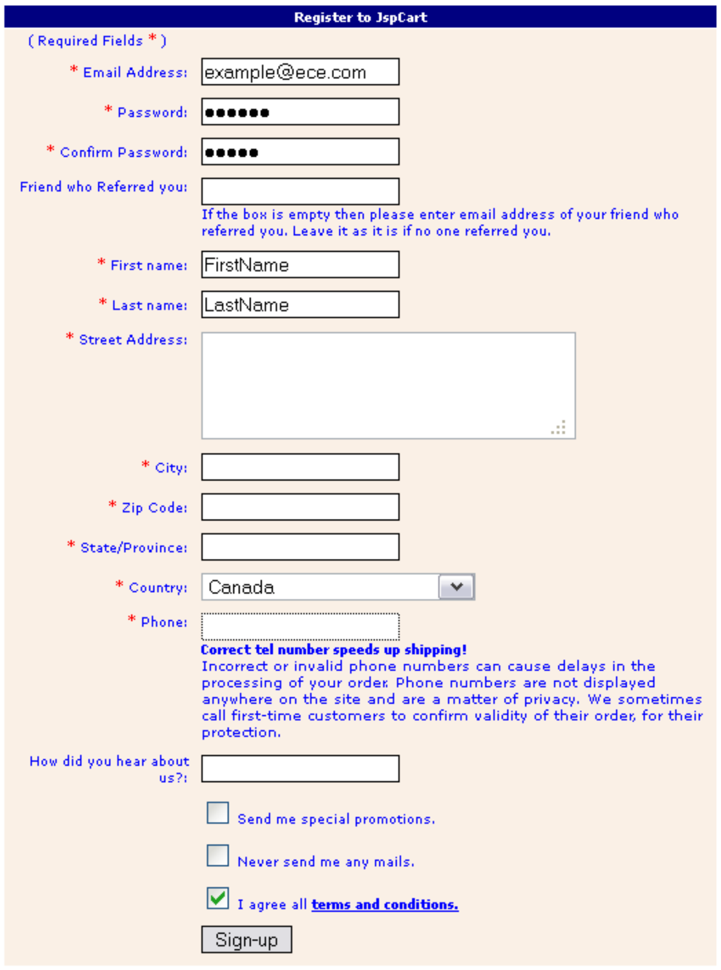
In our example, when the user clicks the "Sign-up" button on the registration page, it triggers an HTTP request to the server side. After that, the server processes the request and inserts all the data from the registration form into the database (jspcart.sql). Subsequently it returns a web page to inform users whether the registration is successful.

[84] http://www.neurospeech.com/Products/JspCart.html
[85] http://tomcat.apache.org/
[86] http://www.mysql.com/

The following code shows the form submission skeletal code. The HTML form has many attributes[87]. The action attribute specifies the URL (Signup.jsp) that accepts the form data when the form is submitted. The name attribute specifies a name (SignupForm) for the form, and the method attribute specifies the HTTP method to transfer the form data (POST). In addition, the HTML form also supports many event attributes. The onsubmit attribute is used to execute custom JavaScript code when the form is submitted.

```
<form action="Signup.jsp" onsubmit="if(verifyForm()) return true;
else return false;" method=POST name="SignupForm">
     …
<input class=DarkButton type=submit name=Submit value="Sign-up">
</form>
```

The issue with the traditional form submission is that the entire web page (which includes all elements) is reloaded every time the submission is triggered. The user has to wait for the web page to be refreshed, which affects the user's experience.



Register to JspCart

(Required Fields *)

* Email Address:	example@ece.com
* Password:	●●●●●●
* Confirm Password:	●●●●●
Friend who Referred you:	
	If the box is empty then please enter email address of your friend who referred you. Leave it as it is if no one referred you.
* First name:	FirstName
* Last name:	LastName
* Street Address:	
* City:	
* Zip Code:	
* State/Province:	
* Country:	Canada
* Phone:	
	Correct tel number speeds up shipping! Incorrect or invalid phone numbers can cause delays in the processing of your order. Phone numbers are not displayed anywhere on the site and are a matter of privacy. We sometimes call first-time customers to confirm validity of their order, for their protection.
How did you hear about us?:	
	☐ Send me special promotions.
	☐ Never send me any mails.
	✓ I agree all **terms and conditions.**
	Sign-up

Figure 7.1 The User Registration Web Page of JspCart

[87] http://www.w3.org/TR/html401/.

7.1.2 Problem 2: Validation

The JavaScript function verifyForm() is created to perform validation for the registration form; however, the validation is invoked on submission. The user has to click the submit button to be notified of the validation results. Additionally, the error message is displayed using alert() which is a modal window. Thus, if an error occurs, the user is redirected between the registration page and the alert message window. In this scenario, the user has to memorize the information being provided in the alert message window, as the error message is lost when this window is closed.

The registration form also has server side validations using JavaBeans (Users.java). The issue with server side validation is that it uses an HTML input element to show the error message on the top of the form. Moreover, the validation applies the classic web application model. That is, the request is sent to the server, and an entire web page is reloaded to display the error message from the server. Furthermore, when an error is encountered, there are no changes to the background or border color of the field with the error.

7.2 Methodology

Web 2.0 forms can avoid these issues easily.

1. Ajax should be used to send asynchronous requests to the server.
2. Both client-side and server-side validations should be performed. The validations should be invoked on change, not on submit. A survey showed that only 22% of forms used validations on change with Ajax (Smashing Magazine, 2008).
3. Clear, unambiguous and visible error messages should be displayed. To get more attention from users, the form should have visual effects (Smashing Magazine, 2008): (1) the error message should be displayed in an attention grabbing color. (2) The background or border color of the input field with the error should be changed. (3) The focus should be altered to the first input field with the error. However, 84% of the web forms did not have visual and focus effects. Furthermore, the error message should be placed in the right place (A survey shows that 57% of such error messages are below the input field, and 26% is on the right side of the field (Smashing Magazine, 2008). Moreover, all the error messages should be displayed at once. It is bad practice to redirect users to another page or utilize an alert message window to inform users of the error. 14% of sites still use JavaScript popups for displaying validation feedback (Smashing Magazine, 2008).

FTT is able to transform traditional forms (using HTML, JavaScript and CSS to present the user interface, JSP as the server side scripting language, and MySQL as the database) into Ajax-enabled forms. FTT uses the jQuery[88] framework to minimize the overhead of this transformation. A recent survey shows that usage of jQuery has steadily grown and it is now more popular than other JavaScript

[88] http://jquery.com/

frameworks[89] such as prototype, dojo and mootools. However, transforming existing applications is not straightforward; manual transformation requires that programmers are knowledgeable about jQuery and the interactions between jQuery, HTML and JSP. Such a process can clearly introduce defects; thus, FTT is required to assist with the form transformation.

Refactoring tools support two automated approaches (Mens & Tourwé, 2004), either fully-automatic (user's interaction is not required) or semi-automatic (user's participation is required). Our tool is semi-automatic as the transformation encompasses functionality that cannot be automatically inferred by a program. Our refactoring adopts the invariants, pre and post-conditions refactoring approach (Opdyke, 1992; Roberts, 1999). The pre-condition requires that the syntax of the HTML and JavaScript code before refactoring is functionally correct and the post-condition requires that the syntax of the modified HTML code and the generated jQuery code after refactoring is functionally correct. To preserve the behavior of the to-be-refactored program, unit testing (Coelho et al., 2006) is usually utilized to ensure that defects are not introduced to the program. However, unit tests designed for traditional forms cannot be applied to Ajax-enabled forms, as unit testing cannot test UI-related code associated with Ajax applications. Hence, our refactoring approach performs validation of the refactored code using two methods. First, formal unit testing is introduced to test the refactored JavaScript code. Second, Selenium[90], a capture-replay tool is used to test whether the two versions of the code are functionally identical even though the UIs for them are no longer identical.

The refactoring process implemented by FTT is as follows.
1. Record the form's submission process using Selenium; and add unit tests to test the behavior of the Ajax call.
2. Transform traditional forms to Ajax-enabled forms.
3. Add both client-side and server-side form validations.
4. Replay the form's submission process recorded using Selenium and execute the unit tests to ensure the transformed code can functionally pass these tests.

7.3 Refactoring Traditional Forms into Ajax-enabled Forms

The functionality of our tool is implemented through three different components; we will now discuss each of them in detail.

7.3.1 Form Submission Transformer

The Form Submission Transformer transforms the HTML and JavaScript code (if JavaScript is used) for the traditional form submission into the jQuery-based code for the Ajax form submission. The transformation process comprises of bad smell detection, code rewriting and code generation.

[89] http://www.google.com/trends?q=prototype+%2C+jquery%2C+YUI%2C+dojo%2C+mootools&ctab=0&geo=all&date=all&sort=0
[90] http://seleniumhq.org/

1. Bad smell detection and code rewriting

The bad smells in our refactoring system are the HTML and JavaScript code (if JavaScript is used) for traditional web forms. Detecting the bad smells by parsing and extracting the HTML and JavaScript code, and rewriting the HTML and JavaScript code are the first two steps in implementing the transformation. Jsoup, a Java HTML parser, is utilized to find, extract and modify HTML elements, attributes and the content of the elements. A JavaScript Parser generated using ANTLR is adopted to find, extract and modify the HTML attributes set by JavaScript.

Bad smell detection includes: (1) extracting the action, method and id attributes specified by the form element by parsing the HTML code. If the action and the method attributes are null in the HTML code, the JavaScript Parser parses the JavaScript code to extract the value. If the setting for the action attribute cannot be found in the JavaScript code, an error will occur because either the settings for the traditional form are incorrect or the form to be refactored is already an Ajax form. If the setting for the method attribute cannot be found in the JavaScript code, the default value (POST) will be used. (2) Extracting the id attribute specified by all the input, textarea and select elements within the <form></form> tag. (3) Extracting the id or class attribute specified by the submit input element.

Bad smell code rewrite contains: (1) setting the action and method attributes to null. The specifications for both of the attributes are to be moved into a jQuery Ajax function, which will be discussed in the following sections. (2) Adding the id attribute to the form element and every input, textarea and select elements within the <form></form> tag, if the id attributes are not specified. This is because the Ajax form submission code uses the jQuery id selector to select these elements. (3) Adding the class attribute to the submit input element. The Ajax form submission code can use the jQuery id or class selector (by default) to choose the submit input element. If both the id and class attributes are not specified, the class attribute is required to be added. (4) Adding JavaScript statements to import the external jQuery plugin. The bad smell detection phase is fully automated; however, the code rewriting phase adopts a semi-automated approach as programmers are required to enter the id and class names for specified HTML elements.

2. Code generation

The third step is to generate the jQuery code for Ajax form submission. JQuery provides a rich set of APIs to develop Ajax applications. jQuery.ajax()[91] is leveraged to perform an asynchronous HTTP request. The following are descriptions of the most frequently used arguments.

- type: specifies the type of the request: GET or POST.
- url: specifies which URL the request is sent to.
- data: specifies which data string is to be sent to the server.
- dataType: specifies which data type is expected to be retrieved from the

[91] http://api.jquery.com/jQuery.ajax/

server. The available types are “xml”, “html”, “script”, “json”, “jsonp”, ‘text” or multiple values.

- success(): the function to be executed after the request is successfully sent to the server.

The code generation phase produces two parts of the code: (1) one part is to retrieve all the form data and to concatenate the data into a string for submission. The string will be served as the value of the data argument in the jQuery.ajax() method. (2) The other part is the jQuery.ajax() method with different arguments. Code generation is invoked in a semi-automatic fashion. Thus, some of the arguments, such as the type and url arguments, are retrieved from the type and url attributes of the form element when parsing the HTML or the JavaScript code (if JavaScript is used). Others are provided manually, such as the dataType argument. Moreover, the generated success() function is a code skeleton. The programmer is required to provide the arguments and the “body” of this function as the system has no way of understanding the domain of the application. To inform the status of submission, the success() function usually takes an argument to get a return value from the server. Hence, the programmer is also required to output a value in the code of the server side scripting to indicate the submission status.

3. The grammar for transformation

In this section, we provide the grammar for the HTML code before and after refactoring, and the grammar for the generated jQuery code for the Ajax form submission. We extended the HTML grammar[92] and JavaScript grammar[93]. To make the grammar simpler, only the directly utilized symbols and rules of the grammar are included.

This is the grammar for the HTML code before refactoring.

```
form: '<form' WS (ATTR)+ '>' (form_field | body_content)* '</form>';
form_field: INPUT | select | textarea;
body_content: body_tag | text;
INPUT: '<input' (WS (ATTR)*)? '>';
select: '<select' (WS (ATTR)*)? '>' (PCDATA)* ('<option' (WS (ATTR)*)? '>' (PCDATA)*)+ '</select>';
textarea: '<textarea' (WS (ATTR)*)? '>' (PCDATA)* '</textarea>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
```

This is the grammar for the HTML code after refactoring.

```
form: '<form' WS 'action' ('=""') 'method' ('=""') ('id' ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))) (ATTR)* '>' (form_field | body_content)* '</form>';
form_field: INPUT | select | textarea;
body_content: body_tag | text;
```

92 http://www.antlr.org/grammar/HTML/html.g

93 http://www.antlr.org/grammar/1206736738015/JavaScript.g

```
INPUT: '<input' (WS ('id' ('=' (WORD ('%')? | ('-')? INT | STRING
| HEXNUM))) | ('class' ('=' (WORD ('%')? | ('-')? INT | STRING |
HEXNUM)))) (ATTR)* '>';
select: '<select' (WS ('id' ('=' (WORD ('%')? | ('-')? INT |
STRING | HEXNUM)))) (ATTR)* '>' (PCDATA)* ('<option' (WS (ATTR)*)?
'>' (PCDATA)*)+ '</select>';
textarea: '<textarea' (WS ('id' ('=' (WORD ('%')? | ('-')? INT |
STRING | HEXNUM)))) (ATTR)* '>' (PCDATA)* '</textarea>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
```

This is the grammar for the generated jQuery code.

```
program: sourceElements EOF;
sourceElements: sourceElement (sourceElement)*;
sourceElement: function | statement;
function: functionDeclaration | jQueryFunction;
jQueryFunction: documentReadyFunction | jQueryCallbackFunction |
ajaxMethod | validationMethod |asyncTestMethod;
documentReadyFunction: '$' '(' 'document' ')' '.' 'ready' '('
'function' '(' ')' '{' sourceElement '}' ')';
jQuerCallbackFunction: '$' '(' selector ')' '.' action '('
(parameter (',' parameter)*)? (',')? callbackFunction ')';
ajaxMethod: '$' '.' 'ajax' '(' '{' ajaxFunctionBody '}' ')';
ajaxFunctionBody: ajaxFuncBody (',' ajaxFuncBody)*;
ajaxFuncBody: ajaxArgument | ajaxFunction;
ajaxArgument: name ':' value;
ajaxFunction: name ':' 'function' '(' arguments (',' arguments)*
')' '{' sourceElements '}';
statement: statementBlock | variableStatement | emptyStatement |
expressionStatement | ifStatement | iterationStatement     |
continueStatement | breakStatement | returnStatement |
withStatement| labelledStatement | switchStatement |
throwStatement | tryStatement;
expressionStatement: expression ';';
expression: assignmentExpression (',' assignmentExpression)*;
assignmentExpression: conditionalExpression | jQueryExpression |
QunitAssertion | leftHandSideExpression assignmentOperator
assignmentExpression;
jQueryExpression: '$' '(' selector ')' '.' action '(' (parameter
(',' parameter)*)? ')';
```

7.3.2 Validation Code Generator

Form validation can be implemented on both client and server side (Mitchell, 2000). Client-side form validation does not require data from the server. It is instantaneous as errors are identified before the form is submitted to the server. Typically, client-side form validation checks whether the required information (such as username and password) is filled and whether the information is in the correct format (such as email address, URL, date and phone number). The user will be informed with correction suggestions if the validation fails. Server-side form validation requires information from the server side. The user's input is sent to the server for validation using a server scripting language and the validation feedback is displayed after the client (browser) receives the server's response. Server-side form validation further checks the information before the data is processed on the server side (such as inserting data into a database) to ensure the correct operation of the web application. In addition, if JavaScript is not enabled in a browser, client-side form validation is disabled; however, server-side form validation is still able to secure the web application. As a result, combining client-side and server-side form validation helps improve the user's experience and provides responsive and secure form validation.

1. The seven steps to add validation

We built a Validation Code Generator to add client and server-side form validation code. The validation code is written using the jQuery validation plugin[94]. This component is also semi-automatic as it requires selections from the programmer. There are seven steps to generate the form validation code.

Step 1: Select validation rules. The jQuery validation plugin provides a list of built-in validation methods to be added to different HTML inputs, text areas and select menus. The first step is for the programmer to provide validation rules for each element in the form. Figure 7.2 shows the selection window in our system for adding validation rules. One exception is the validation rule for "Maxlength". The input elements have an attribute called maxlength, if this attribute is specified in the HTML code; the value of the attribute is retrieved during the HTML code parsing and will be automatically displayed in Figure 7.2. Additionally, if server-side validation (database involved) is required, the programmer specifies the URL of a server side scripting file in the "Remote" field. All the built-in validation methods are common rules for validation, custom validation methods can be added by clicking the "Add Rules" button. Both client-side and server-side validation rules can be added using jQuery.validator.addMethod(). Furthermore, the jQuery validation plugin also provides more complex validation methods in additional-methods.js, such as the validation rules for time, phone number and IP address. Hence, if validation rules other than the basic rules are required, the programmer can click the "More Rules" button for additional options.

Step 2: Server side validation code setting. If a programmer specifies a URL for the server side validation, they are required to provide the server name, database name, table name, username and password for the database, to generate the server side scripting file.

Step 3: Specify error messages. The jQuery validation plugin provides default error messages for different validation methods. For example, the error message is "This field is required", if this field is left empty. Custom error messages are also available. Thus, this step is for the programmer to specify the custom error messages for different form fields. This step is optional.

Step 4: Select the display position for error messages. The options include above or below the field and on the right or left side of the field. This step is optional.

[94] http://bassistance.de/jquery-plugins/jquery-plugin-validation/

Step 5: Select the style of error messages. FTT provides different CSS style of error messages. The programmer can select the style that is accordance with the style of the form presentation. This step is optional.

Step 6: Code generation. The system generates the jQuery form validation code in line with the programmer's requirements. The validation code can be placed into the input, textarea and select elements using the class attribute or be placed into the $(document).ready function which occurs when the DOM has been loaded. If the programmer specifies a URL for the server side validation, a server side scripting file whose name is indicated by the specified URL in the "Remote" field is generated using a JSP code template. The JSP code returns true or false to the client side to indicate the validation status. Subsequently, form users are informed as soon as the client side obtains the validation status.

Step 7: Clean up. Traditional web forms before refactoring perform validations using custom JavaScript functions. The final step is to clean up these JavaScript functions and their specifications.

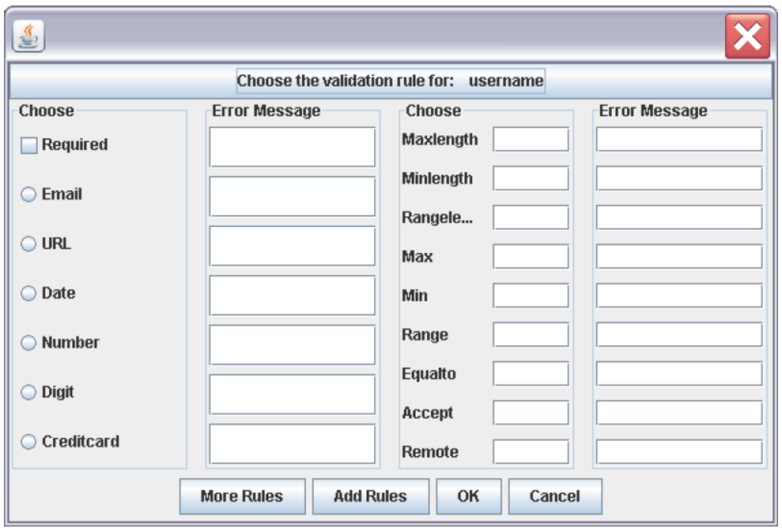


Figure 7.2 The Validation Selection Window

2. The grammar for generated form validation code

Two parts of the code are generated: the JavaScript code for importing external JavaScript libraries, and the validation code using the jQuery validation plugin. The following code shows the grammar for the generated validation code if the programmer selects to place the validation code into the HTML elements and the grammar for the generated validation code if the programmer selects to placed the validation code into the $(document).ready function. As described in the previous section, we only include the directly utilized symbols and rules of the grammar.

This is the grammar for the generated validation code placed into HTML elements.

```
form: '<form' WS 'action' ('=""') 'method' ('=""') ('id' ('='
(WORD ('%')? | ('-')? INT | STRING | HEXNUM)))(ATTR)* '>'
(form_field | body_content)* '</form>';
form_field: INPUT | select | textarea;
body_content: body_tag | text;
INPUT: '<input' (WS ('id' ('=' (WORD ('%')? | ('-')? INT | STRING
| HEXNUM))) | ('class' '=' validationRules)) (ATTR)* '>';
select: '<select' (WS ('id' ('=' (WORD ('%')? | ('-')? INT |
STRING | HEXNUM))) | ('class' '=' validationRules)) (ATTR)* '>'
(PCDATA)* ('<option' (WS (ATTR)*)? '>' (PCDATA)*)+ '</select>';
textarea: '<textarea' (WS ('id' ('=' (WORD ('%')? | ('-')? INT |
STRING | HEXNUM))) | ('class' '=' validationRules)) (ATTR)* '>'
(PCDATA)* '</textarea>';
ATTR: WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
validationRules: '"{' rules (',' message)? '"}';
rules: rule (',' rule)*;
message: 'messages' ':' '{' rule (',' rule)* '}';
rule: ruleName ':' value;
```

This is the grammar for the generated validation code placed into the $(document).ready function.

```
validationMethod: '$' '(' formID ')' '.' 'validate' '(' '{'
validateBody '}' ')';
validateBody: options (',' options)*;
options: rule | callbackFunc | option;
callbackFunc: funcName ':' 'function' '(' parameter (','
parameter)* ')' '{' sourceElements '}';
option: optionName ':' '{' input (',' input)* '}'
optionName: 'rules' | 'message';
input: inputName ':' (('{' rule (',' rule)* '}') | value);
rule: ruleName ':' value;
```

7.3.3 QUnit Code Generator

Ajax calls are essential to Ajax forms. To provide programmers with a mechanism to test Ajax calls after refactoring, a QUnit Code Generator is built to generate multiple test cases. Unit testing is usually synchronous and the test cases are executed one after another. To test asynchronous functions, such as Ajax calls or functions called by setTimeout() or setInterval(), in the test() method, stop() is called before asynchronous operation, and start() is used after all assertions are called (Lindley, 2009). QUnit[95], a powerful JavaScript test suite, provides the asyncTest() method to test Ajax calls. It is a shortcut as it is equivalent to calling the test() and stop() methods. Thus, the generated code utilizes asyncTest() to implement the asynchronous testing.

[95] http://docs.jquery.com/Qunit

Ideally, this tool should be utilized as the first phase in the refactoring process to validate that the new Ajax-based form successfully implements the original HTML-based form. It is a semi-automated tool, which requires programmers to provide information for the testing code generation. This information includes: the name of the test case, the name of the variable to be tested, the expected value of the test case and the URL to send the Ajax request to. To test the Ajax call for the form submission, a URL is set to the server side scripting file where all the form data is submitted to. Similarly, to test the Ajax call for server-side validation; a URL is set to the server side scripting file generated by the Validation Code Generator.

The following code shows the grammar for the generated QUnit testing code. As described in the previous section, we only include the directly utilized symbols and rules of the grammar.

```
program: sourceElements EOF;
sourceElements: sourceElement (sourceElement)*;
sourceElement: function | statement;
function: functionDeclaration | jQueryFunction;
jQueryFunction: documentReadyFunction | jQueryCallbackFunction |
ajaxMethod | validationMethod |asyncTestMethod;
asyncTestMethod: asyncTest (';' asyncTest)*;
asyncTest: 'asyncTest' '(' testCaseName ',' 'function' '(' ')' '{'
sourceElement '}' ')';
statement: statementBlock | variableStatement | emptyStatement |
expressionStatement | ifStatement | iterationStatement     |
continueStatement | breakStatement | returnStatement |
withStatement| labelledStatement | switchStatement |
throwStatement | tryStatement;
expressionStatement: expression ';';
expression: assignmentExpression (',' assignmentExpression)*;
assignmentExpression: conditionalExpression | jQueryExpression |
QunitAssertion | leftHandSideExpression assignmentOperator
assignmentExpression;
QunitAssertion: ok | equal | notEqual | deepEqual | notDeepEqual |
strictEqual | notStrictEqual | raises;
```

7.4 Transforming the Example Form into an Ajax Form

In Section 7.1, we introduced JspCart's traditional HTML registration form. In this section, FTT is used to transform it into its Ajax-enabled equivalent form.

7.4.1 Record the Form's Submission Process

Before the transformation process, we use Selenium to capture the user's inputs to the registration form and the outputs. After refactoring, we replay the user's interaction to check whether the outputs of the form are unchanged (even though the GUI may have changed), to ensure the functionality of the form is preserved before and after refactoring. For example, we record the process of a successful form submission, and Figure 7.3 shows the output of the successful form submission before refactoring. The browser will be redirected to the Login.jsp if the registration is successful.

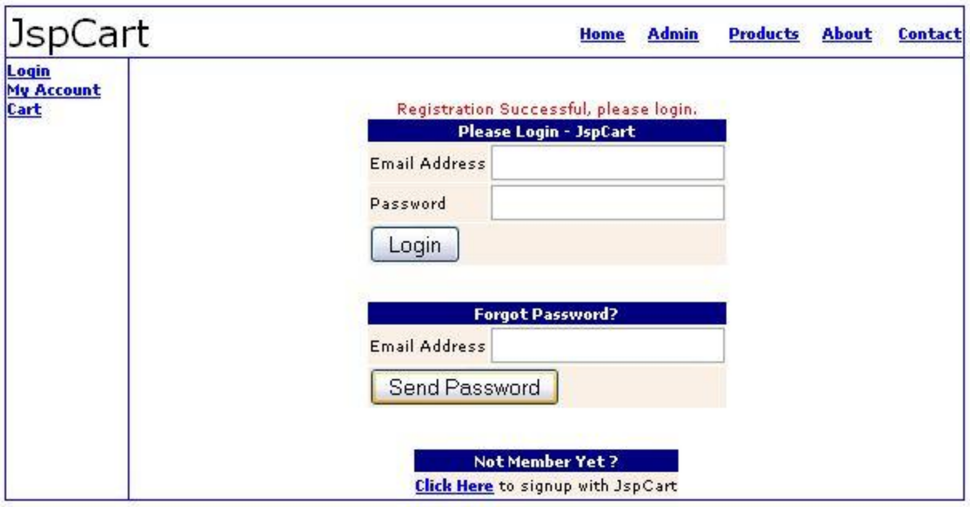


Figure 7.3 The Output of the Form Submission before Refactoring

7.4.2 Add Test Case

To make sure the refactoring is error-free; we add test cases to test Ajax calls for server-side validations (programmers can also add test cases to test the Ajax call for the form submission). If we want to test the Ajax call for validating the email address, we would create the following test unit case. Name: "Test Email Address", test field: “txtEmailAddress”, test value: “example@ece.com”, expected return value: “false” and return message: “The Email is in the database”. We also provide the URL for the server side scripting file (validation.jsp), which will be generated by our Validation Code Generator. The generated QUnit code by the QUnit Code Generator is shown as follows.

```
asyncTest("Test Email Address", function(){
   setTimeout(function(){
      var dataString = 'txtEmailAddress = '+' example@ece.com';
      $.ajax({
         url: "validation.jsp",
         data: dataString,
         success: function(response) {
              equals(response, "false", "The Email is in the
              database!");
              start();
         }
      });
   }, 100)
});
```

7.4.3 Transform the Registration Form into an Ajax Form

The next step is to transform the traditional form submission method through the Form Submission Transformer using the process below.

HTML code detection and rewriting: The Form Submission Transformer performs the following modifications to the code. (1) The action and method attributes of the form are set to null. (2) The id attribute is added. (3) JavaScript statements to import the external jQuery plugin are added.

JQuery code generation: The Form Submission Transformer then generates the Ajax submission code. The generated code for the basic configuration is shown as follows.

```
$(".DarkButton").click(function(){
   var txtEmailAddress = $("#txtEmailAddress").val();
   var txtFirstname = $("#txtFirstname").val();
   var txtLastname = $("#txtLastname").val();
   …
   var dataString = "txtEmailAddress = " + txtEmailAddress +
      "&txtFirstname = " + txtFirstname + "& txtLastname = " +
      txtLastname + …;
    $.ajax({
      type: "POST",
      url: "Signup.jsp",
      data: dataString,
      dataType: "html",
      success: function(data){
         if(data == 1){
            location.href = "Login.jsp?txtMessage = Registration
               Successful,%20please%20login.&txtRedirect    =    "    +
               '<%=org.nspeech.web.UrlEncoderEx.encode(txtRedirect)%>
               ';
         }
        return false;
      }
    });
});
```

The “DarkButton” is the class name of the submit button. When the submit button is clicked, the input information is gathered using the jQuery id selector and concatenated into a string “dataString”. In the jQuery.ajax() method, the type of the request, the URL to send the request to, the data type and the data to be sent to the server are specified (The user provides the data type information). The other information is retrieved by HTML parsing and the generated code (dataString). In addition, a success() function, which is called when the request is successfully send to the server, is generated with an empty function body. The final step is for programmers to add any domain specific component to this function.

According to the system flow, after the user has successfully registered, the web page will be redirected to the Login.jsp for the user to login. Thus, the redirection code is moved from the JSP snippet in the Signup.jsp to the success() function. However, a parameter (data) is required for the success() function to inform that the registration is successful before redirecting the user to the login page. The data type for the parameter returned from the server is “html”. Meanwhile, we output a number “1” after the statements of the successful database operation in the JSP snippet of the Signup.jsp.

7.4.4 Add Validations

There are seven steps used to generate the form validation code.

Step 1: Select the validation rules. We select different validation rules for different fields. For the email field, we also specify the URL for the server side scripting file (validation.jsp) containing the server-side validation code.

Step 2: Server-side validation code setting. We provide the server name, database name, table name, user name and password for the database to generate the server side scripting file (validation.jsp).

Step 3: Specify the error messages. We specify the error messages for the email and the confirm password field. If the user’s email has been registered previously, the error message “The Email has already been registered!” will be displayed (as can be seen in Figure 7.4). Moreover, if the two passwords do not match, the error message “Two passwords must match!” will be shown.

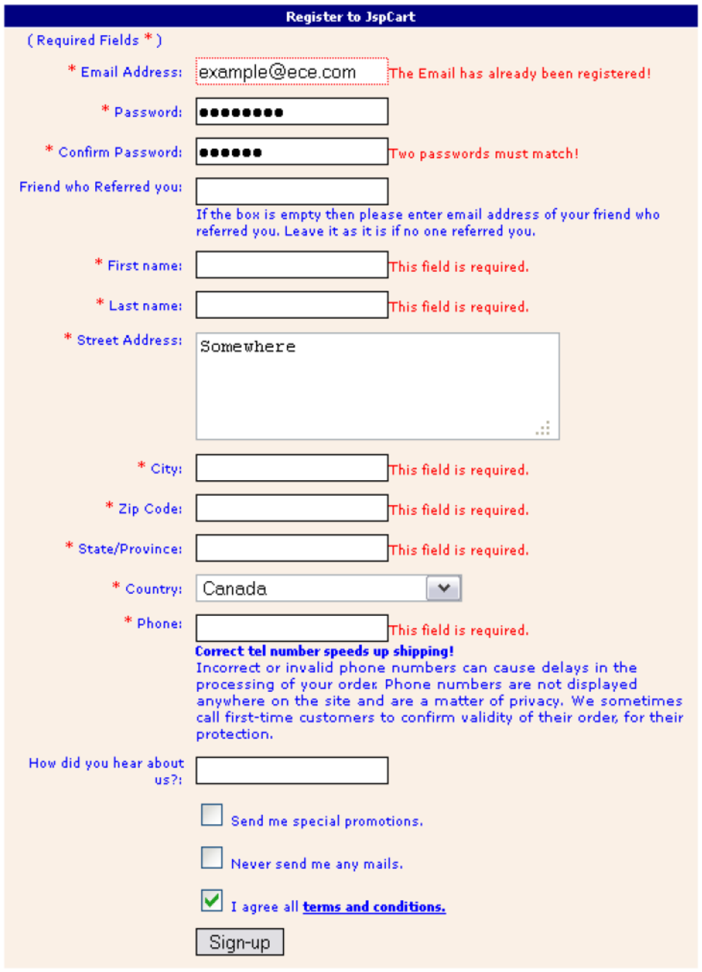
Step 4: Select the position of the error messages. We place the error message on the right side of the field.

Step 5: Select the style of the error messages. For our style, the error message is displayed in red and the border of the first input field with an error becomes dotted and red to attract the user’s attention.

Step 6: Code generation. The system generates the validation code and the server side scripting file (validation.jsp) in line with our requirements. The validation code is placed into the $(document).ready function.

Step 7: Clean up. We delete all the JavaScript functions for the client and server-side validation and we also delete the onsubmit attribute specified by the form element.

Figure 7.4 shows the registration form after the validations have been added.



Register to JspCart

(Required Fields *)

* Email Address:	example@ece.com	The Email has already been registered!
* Password:	●●●●●●●●●	
* Confirm Password:	●●●●●●	Two passwords must match!
Friend who Referred you:		
	If the box is empty then please enter email address of your friend who referred you. Leave it as it is if no one referred you.	
* First name:		This field is required.
* Last name:		This field is required.
* Street Address:	Somewhere	
* City:		This field is required.
* Zip Code:		This field is required.
* State/Province:		This field is required.
* Country:	Canada	
* Phone:		This field is required.
	Correct tel number speeds up shipping! Incorrect or invalid phone numbers can cause delays in the processing of your order. Phone numbers are not displayed anywhere on the site and are a matter of privacy. We sometimes call first-time customers to confirm validity of their order, for their protection.	
How did you hear about us?:		

☐ Send me special promotions.

☐ Never send me any mails.

☑ I agree all **terms and conditions.**

Sign-up

Figure 7.4 The Registration Form after Refactoring

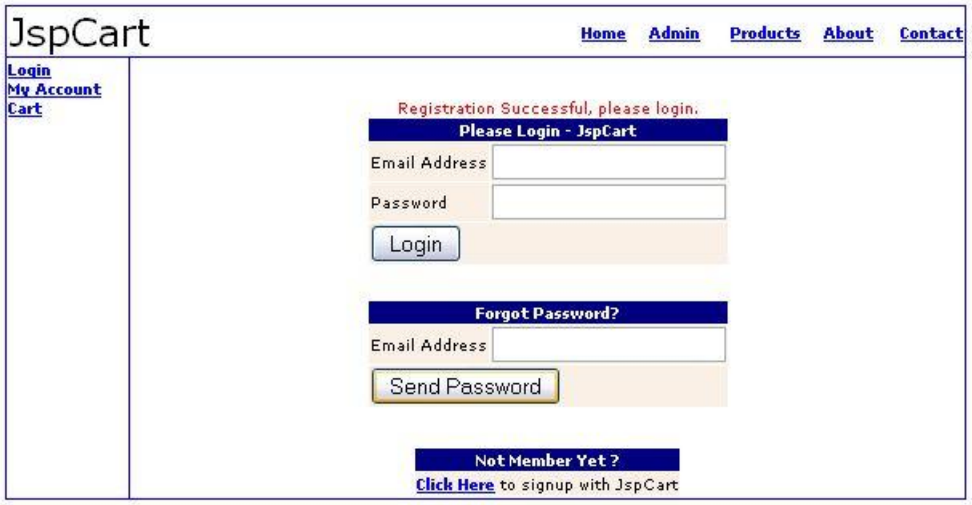
The following code shows the generated jQuery validation code. The “SignupForm” is the id of the registration form and the validate() is the method to validate the selected form. The “rules” and “Message” are options of the validate() method, which are used to specify all the validation rules and the custom messages through key/value pairs.

```
$("#SignupForm").validate({
   rules:{
      txtEmailAddress:{
         required: true,
         email: true,
         remote: "validation.jsp"
      },
      txtPassword:{
         required: true,
         minlength: 5
      },
      txtPassword2:{
        required: true,
        minlength: 5,
        equalTo: "#txtPassword"
      },
     …
    },
    messages:{
      txtEmailAddress:{
        remote: "The Email has already been registered!"
      },
      txtPassword2:{
        equalTo: "Two passwords must match!"
      }
    }
})
```

7.4.5 Replay the Form's Submission Process and Execute the Unit Test

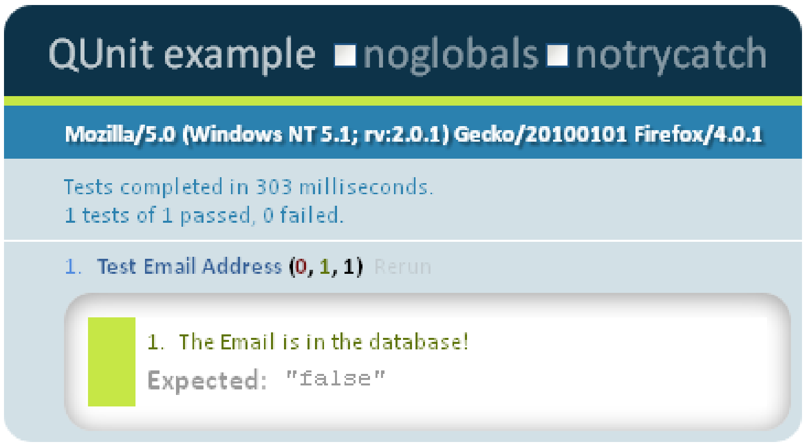
After refactoring, we replay the Selenium script that recorded the user's interaction of a successful submission, and the form reproduces the output without anomalies as shown in Figure 7.5. This output is the same as the output before refactoring, which implies that our refactoring does not change the functionality of the form.

In the previous sections, we add the test case to test the Ajax call which checks whether the email address "example@ece.com" has already been registered. Thus, after refactoring, we execute the test code to see whether our generated validation code passes the test case. Figure 7.6 shows the test passes. The test case returns "false" which means the specified email address is in the database of the JspCart web application.



Registration Successful, please login.

Figure 7.5 The Output of the Form Submission after Refactoring



Mozilla/5.0 (Windows NT 5.1; rv:2.0.1) Gecko/20100101 Firefox/4.0.1

Tests completed in 303 milliseconds.
1 tests of 1 passed, 0 failed.

1. Test Email Address (0, 1, 1) Rerun

1. The Email is in the database!
Expected: "false"

Figure 7.6 The QUnit Testing Result

7.4.6 Case Study 2

Petstore[96], an open source e-commerce application, will also be transformed. Petstore has several forms in its website management system; we will take the form for adding products information as an example (productmanager.jsp). The refactoring process transforms the traditional form for adding products' information into an Ajax version. FTT modifies the following HTML code (1) the action and method attributes of the form are set to null. (2) The id attributes are added. (3) The class attribute of the submit input element is added. (4) JavaScript statements to import the external jQuery plugin are added. FTT then generates the Ajax submission code which includes: (1) the code to retrieve information for "Product Name", "Uniprice" and "Category" from the HTML elements, and to concatenate all the data into a string. (2) The code for the jQuery.ajax() method is also produced with different arguments. No matter whether the submission is successful, the browser will be redirected back to the productmanager.jsp. Ajax-enabled forms can prevent needless redirections and can display the status of the submission directly on the productmanager.jsp. Thus, to refactor this form into an Ajax-enabled version, we have added a parameter to the success() function to retrieve the status of the submission from the admin/productadmin. A status message is added to the productmanager.jsp to inform the user about the status of the submission, and the redirection code in admin/productadmin is removed. This way, after the submission, the browser will stay in the productmanager.jsp without redirection.

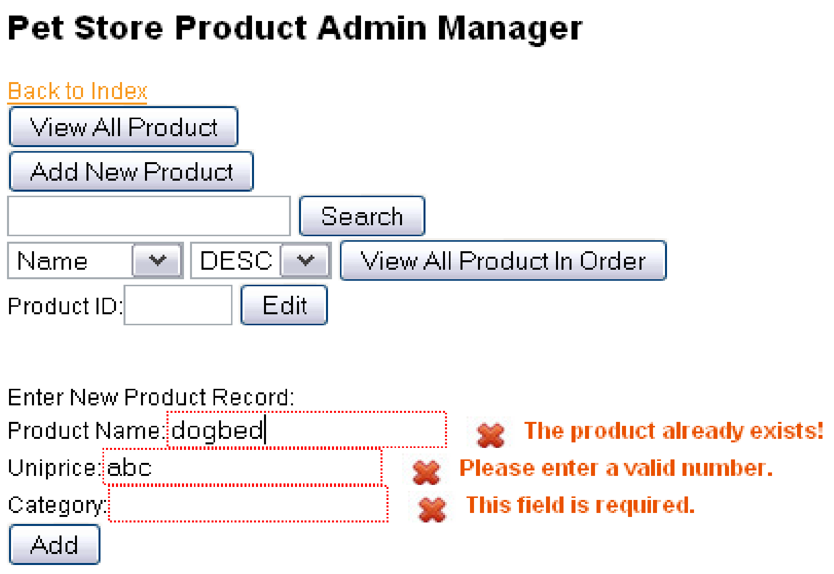
The next step is to add validations to the form. We specify the rule "required" and "productNameCheck" for the Product Name field. The rule "productNameCheck" is a custom server side validation rule to check whether the product already exists in the database. We provide information for FTT to create this rule using the jQuery.validator.addMethod(). In addition, we specify the rule "required" and "number" for the Uniprice field and the rule "required" for the Category field. We

[96] http://code.google.com/p/petstorewebsite/

also specify the error message for the newly created rule “productNameCheck”. All the error messages are selected to display in red with an icon in front and the border of all the input fields with errors are dotted and red. The generated validation code is selected to be placed into the HTML elements using the class attribute. To use the class attribute for adding validations, the JavaScript statement to import jquery.metadata.js is generated at the same time. The following code shows the generated validation code for the form used to add products information in the product management page.

```
<form action = "" name = "newproductrecord" method = ""
   id = "newproductrecord">
   <input type = "hidden" name = "mode" id = "mode"
     value = "<%=addnewmode%>"/>
   Product Name: <input type = "text" name = "pname" id = "pname"
     value = ""  size = "20" class = "{required:true,
     productNameCheck:ture,messages:{productNameCheck:'The product
     already  exists!'}}"/><BR>
   Uniprice: <input type = "text" name = "uniprice" id =
     "uniprice" value = "" size = "20" class = "{ required:true,
      number:true}" /><BR>
   Category: <input type = "text" name = "category" id =
     "category" value = "" size = "20" class = "{required:true}"
     /><BR>
    <input type = "submit" value = "Add" name = "addnew" class =
      "submitform"/>
</form>
```

Figure 7.7 shows the product management page after refactoring, which demonstrates that the transformation is performed successfully.


Pet Store Product Admin Manager

Back to Index

Enter New Product Record:
Product Name: dogbed
The product already exists!
Uniprice: abc
Please enter a valid number.
Category:
This field is required.

Figure 7.7 The Product Management Page after Refactoring

After the transformation, we count the number of lines of code (LOC) generated by FTT (not including comment lines, blank lines, lines of a single brace or parenthesis) to evaluate the effort saved by programmers. Table 7.1 shows the number of LOC generated for the form submission, form validation and test cases (using Qunit) for the two web applications (JspCart and Petstore). The number of LOC generated to perform form validation includes the code for server side validation. The number of LOC generated for form submission, form validation and test cases are dependent on how many fields are in the form, how many fields requiring validation and how many test cases are required to produce the desired coverage. JspCart has 17 fields (including one hidden field), 12 fields requiring validation and 1 test case. Petstore has 4 fields (including one hidden field), 3 fields requiring validation and 1 test case.

Table 7.1 The Number of Lines of Code Generated by FTT

Application	Form Submission(LOC)	Validation(LOC)	Test Cases(LOC)
JspCart	33	97	28
Petstore	16	58	28

7.5 Evaluation

We tested the efficiency improvement of the two forms in the JspCart and Petstore to evaluate the effectiveness of the refactored systems. We measure the "response time" before and after refactoring as an amalgamation of the network transfer time, the network latency time, the server response time and the client processing time.

For JspCart, we test the response time of the server side validation for the form field "Email Address" when the server side validation fails. Before refactoring, the entire form needs to be submitted before the field can be validated, so the time measurement starts from when the user clicks on the Sign-up button and ends when the server side returns the error message to the registration page. After refactoring, validation is done on change, so the time measurement starts from when the user finishes entering the email address to trigger an XMLHttpRequest request and ends when the error message is displayed on the right side of the "Email Address" field.

For Petstore, we test the response time of the form submission before and after refactoring. Before refactoring, the time measurement starts from when the user clicks on the submission button on the productmanager.jsp page and ends when the browser is redirected back to the productmanager.jsp page. After refactoring, the time measurement starts from when the user clicks on the submission button and ends when the status of submission is displayed on the productmanager.jsp page.

Tests were executed in the following environment: Intel(R) Core (TM) 2 Quad CPU Q6600 @2.4GHz, with 4 GB of RAM running Microsoft Windows XP

Professional, Service Pack 3. In addition, each test was executed 100 times in Firefox 4.0 and an average was taken to ensure that any extraneous timing issues were minimized.

Table 7.2 shows the response time before and after refactoring and their relative performance is compared. The table shows that, for both applications, the response time for the refactored (Ajax-enabled) forms are significantly faster than the original (traditional) forms. The relative mean difference (RMD) of the response time before and after is calculated as:

$$\frac{\textit{(Response Time before Refactoring - Response Time after Refactoring)}}{\textit{(Response Time before Refactoring + Response Time after Refactoring) /2}} \qquad (4)$$

Table 7.2 Testing Results for the Response Time before and after Refactoring

Application	Response time before Refactoring (ms)	Response time after Refactoring (ms)	RMD
JspCart	119.5	15.8	1.53
PetStore	202.1	46.9	1.25

Chapter 8 Conclusion

As functionality, especially the interactive functionality, increases in RIAs, performance becomes a more significant issue. This issue is compounded by the movement of such applications onto mobile platforms such as smartphones and tablets. Transforming poor performing RIA code into more efficient code is not a straightforward process. If left solely to the programmer, the result will often be produce new structures that contain defects while not improving the efficiency of the system significantly. This dissertation proposes refactoring as a technique to help improve the efficiency of these RIAs.

Chapter 4 introduces refactoring to help remove bad smells in AS3 code to improve users' experiences by making AS3 code run faster. To avoid the tedious, error and omission-prone manual refactoring process, this chapter proposes a refactoring tool, ART, which automatically produces refactorings with minimal programmer intervention. An empirical study demonstrates that ART produces significantly faster code. While this system explicitly targets ActionScript, migrating the system to support other languages derived from ECMA-262 Script v3-5 is relatively straightforward.

Chapter 5 proposes a technique to better embed Flash content into web pages. Markup-based embedding methods or JavaScript-based embedding methods can be used to include Flash content; however, JavaScript-based embedding methods result in superior implementations. This chapter introduces a refactoring tool called FlashembedRT to assist programmers transform any of the five markup-based embedding methods into a method using one of the popular Flash embedding JavaScript libraries, flashembed. An example is provided to demonstrate how programmers can use FlashembedRT to refactor their Flash embedding methods.

Chapter 6 discusses refactoring as a method to modify existing RIAs to use JSON instead of XML for the data exchange format. Due to its lightweight nature and native support for JavaScript, JSON, an alternative data exchange format to XML, improves the efficiency of Ajax applications. Specifically, it improves the efficiency with respect to (1) network transfer time, and (2) JavaScript processing time on the client side. Changing the data format for an existing Ajax application involves: (1) converting the data format from XML to JSON; and (2) changing the JavaScript code – from code which manipulates the XML version of the data to code which manipulates the JSON version of the data.

A tool called XtoJ is introduced to aid the programmers with the transformation from XML to JSON. This system is based around three components: XML to JSON Converter, JavaScript Code Transformer and JavaScript Code Generator. An empirical demonstration shows that XtoJ can significantly improve the efficiency of Ajax applications; and the improvements are consistent with efficiency reports for manual construction (Nurseitov et al., 2009). An analysis is also performed on three additional variables that can influence the performance of the system (the number of objects, the structure of the XML and JSON data and the length of the text in a node). These additional results imply that the benefits of transforming such systems will significantly improve a large number of applications.

Chapter 7 presents a system called FTT to refactor traditional web forms into Ajax-enabled web forms. Using traditional forms can lead to a poor user experience because the entire web page is reloaded each time the form is submitted. Furthermore, form validations on both the client side and the server side are triggered upon submission, not on change. This model leads to inefficiency in both responsiveness and data transmission. Hence, traditional forms should be refactored into Ajax-enabled forms. The system contains three components: Form Submission Transformer, Validation Code Generator and QUnit Code Generator. The Form Submission Transformer refactors the original HTML and JavaScript code into the jQuery-based code. The Validation Code Generator aids the programmer with client-side and server-side form validation by automatically generating the validation code based on inputs from the programmer. Finally, the QUnit Code Generator helps programmers generate code for unit testing to ensure the form still meets the system requirements after the refactoring process. Two case studies are performed to demonstrate the capabilities of the system. These show that FTT can be used to correctly refactor traditional forms into Ajax-enabled forms.

8.1 Future Work

Exploring additional issues and expanding the capabilities of the refactoring tools introduced in this dissertation form the basis for future work.

Currently the presented tools are static refactoring tools with minimal user overhead; therefore, future works include adding dynamic features, which can pass performance and improvement information to programmers and improve the communication with them. However, such extensions require great flexibility because different programmers have differing tolerances for overheads. Hence, an interactive system must allow the programmers to select the “amount” (or “level”) of overhead (or “interaction”) which they are willing to tolerate. Without such flexibility, experience has shown that the support system will quickly be abandoned by the programming community.

Although programmers can use the proposed refactoring systems to improve their web applications, the systems are not well integrated with each other. That is, programmers have to use multiple tools if they want to improve various aspects of their applications. For example, they have to use XtoJ and FTT to improve the data format and to refactor their forms into Ajax-enabled forms. Additional work will concentrate on integrating all the proposed refactoring systems into a fully comprehensive tool. This tool can then assist programmers improve the efficiency of their web applications through either optimization techniques (ART, XtoJ) or migration of their systems to Ajax-enabled versions (FTT). Finally, the tool will be released to the public. Once the tool is released, a survey can be done from the programmers using the tool to determine the effectiveness of the tool using a much larger sample size with a variety of different configurations (Eaton et al. 2007).

Bibliography

Ali, A.D. (2001). A Dynamic Cluster Constructor for Load Balancing in Big Heterogeneous Distributed Systems. Proceedings of the Symposium on Performance Evaluation of Computer and Telecommunication Systems, pp. 47-55.

Allen, R., Qian, K., Tao, L., Fu, X. (2008). Web Development with JavaScript and Ajax Illuminated. Sudbury, MA, USA: Jones & Bartlett Learning.

Allsopp, J. (2009). Developing with Web Standards. New Riders: Berkeley, CA, USA.

Amiri, K., Park, S., Tewari, R., Padmanabhan, S. (2003). DBProxy: A Dynamic Data Cache for Web Applications. Proceedings of the 19th International Conference on Data Engineering. IEEE Computer Society: Washington, DC, USA, pp. 821-831.

Arnold, M., Fink, S., Grove, D., Hind, M., Sweeney, P.F. (2000). Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model. Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.

Attenborough, M. (2003). Mathematics for Electrical Engineering and Computing Electronics & Electrical. Boston, MA, USA: Newne.

Babic, D., Rakamaric, Z. (2002). Bytecode Optimization. Proceedings of the 24th International Conference on Information Technology Interfaces, pp. 377-383.

Bacon, D.F., Graham, S.L., Sharp, O.J. (1994). Compiler Transformations for High-performance Computing. ACM Computing Surveys (CSUR) 26(4): 345-420.

Bala, V., Duesterwald, E., Banerjia, S. (2000). Dynamo: A Transparent Dynamic Optimization System. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM: New York, NY, USA. 35(5), pp. 1-12.

Balasubramanian, J., Schmidt, D.C., Dowdy, L., Othman, O. (2004). Evaluating the Performance of Middleware Load Balancing Strategies. Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference. IEEE Computer Society: Washington, DC, USA, pp. 135-146.

Bartolini, S., Prete, C. (2005). Optimizing Instruction Cache Performance of Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS) 4(4), pp. 934-965.

Bergeron, J., Debbabi, M., Erhioui, M.M., Ktari, B. (1999). Static Analysis of Binary Code to Isolate Malicious Behaviors. Proceedings of the 8th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 184-189.

Bergsten, H. (2000). JavaServer Pages. Sebastopol, CA, USA: O'Reilly Media.

Beyls, K., D'Hollander, E.H. (2009). Refactoring for Data Locality. IEEE Computer 42(2), pp. 62-71.

Braunstein, R. (2007). Introduction to Flex 2. Sebastopol, CA, USA: O'Reilly Media.

Braunstein, R., Wright, M.H., Noble J.J. (2007). ActionScript 3.0 Bible. New York, NY, USA: Wiley.

Buck, B., Hollingsworth, J.K. (2000). An API for Runtime Code Patching. International Journal of High Performance Computing Applications 14(4), pp. 317-329.

Bulka, D., Mayhew, D. (2000). Efficient C++: Performance Programming Techniques. Boston, MA, USA: Addison Wesley.

Buyukozkan, G. (2009). Determining the Mobile Commerce User Requirements Using an Analytic Approach. Computer Standards and Interfaces 31(1), pp. 144-152.

Card, S.K., Robertson, G.G., Mackinlay, J.D. (1991). The Information Visualizer, An Information Workspace. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'91), pp. 181-188.

Carey, P. (2009). New Perspectives on HTML, XHTML, and Dynamic HTML. Florence, KY, Cengage Learning.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F.P., Temam, O. (2007). Rapidly Selecting Good Compiler Optimizations using Performance Counters. Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society: Washington, DC, USA, pp. 185-197.

Challenger, J.R., Dantzig, P., Iyengar, A., Squillante, M.S. Zhang, L. (2004). Efficiently Serving Dynamic Data at Highly Accessed Web Sites. IEEE/ACM Transactions on Networking 12(2), pp. 233-246.

Chen, C.W., Chang, C.H., Ku, C.J. (2005). A Low Power-Consuming Embedded System Design by Reducing Memory Access Frequencies. IEICE Transactions E88-D(12), pp. 2748-2756.

Chen, W.K., Lerner, S., Chaiken, R., Gillies, D.M. (2000). Mojo: A Dynamic Optimization System. Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization.

Chu, J., Dean, T. (2008). Automated Migration of List Based JSP Web Pages to AJAX. Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE Computer Society: Los Alamitos, CA, USA, pp. 217-226.

Cierniak, M., Li, W. (1997). Briki: an Optimizing Java Compiler. Proceedings of the 42nd IEEE International Computer Conference. IEEE Computer Society: Washington, DC, USA, pp.179-184.

Cinnéide, M.O., Boyle, D., Moghadam, I.H. (2011). Automated Refactoring for Testability. Proceedings of the 4h IEEE International Conference on Software Testing, Verification and Validation Workshops. IEEE Computer Society: Washington, DC, USA, pp. 437–443.

Coelho, R., Kulesza, U., Staa, A.V., Lucena, C. (2006). Unit Testing in Multi-Agent Systems using Mock Agents and Aspects. Proceedings of the International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. ACM: New York, NY, USA, pp. 83-90.

Cowan, N. (2000). The Magical Number 4 in Short-term Memory: A Reconsideration of Mental Storage Capacity. Behavioral and Brain Sciences 24, pp. 87-185.

Datta, A., Dutta, K., Thomas, H.M., VanderMeer, D.E., Ramamritham, K., Fishman, D. (2001). A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. Proceedings of the 27th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers: San Francisco, CA, USA, pp. 667-670.

Demeyer, S. (2002). Maintainability versus Performance: What's the Effect of Introducing Polymorphism? Technical Report, Universiteit Antwerpe.

Derezińska, A., Sarba, K. (2010). Distributed Environment Integrating Tools for Software Testing. Advanced Techniques in Computing Sciences and Software Engineering, pp. 545-550.

Desoli, G., Mateev, N., Duesterwald, E., Faraboschi, P., Fisher, J.A. (2002). DELI: A New Run-Time Control Point. Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 257-268.

Dig, D. (2011). A Refactoring Approach to Parallelism. IEEE Software 28(1), pp. 17-22.

Dig, D., Marrero, J., Ernst, M.D. (2009a). Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society: Washington, DC, pp. 397-407.

Dig, D., Tarce, M., Radoi, C., Minea, M, Johnson, R. (2009b). Relooper: Refactoring for Loop Parallelism in Java. Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. ACM: New York, NY, USA, pp. 793-794.

Driver, M., Valdes, R., Phifer, G. (2005). Rich Internet Applications Are the Next Evolution of the Web.Technical Report, Gartner.

Dudziak, T., Wloka, J. (2002). Tool-supported Discovery and Refactoring of Structural Weaknesses in Code. M.S. Thesis, Technical University of Berlin.

Eaton, C., Memon, A.M. (2007). An Empirical Approach to Testing Web Applications Across Diverse Client Platform Configurations. International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering 3(3), pp. 227-253.

Eichorn, J. (2007). Understanding AJAX: Using JavaScript to Create Rich Internet Applications. New Jersey, USA: Prentice Hall.

Englander, R. (1997). Developing Java Beans. Sebastopol, CA, USA: O'Reilly Media.

Faris, T.H. (2006). Safe and Sound Software: Creating an Efficient and Effective Quality System for Software Medical Device Organizations. Milwaukee, USA: ASQ Quality Press.

Florio, C., Adobe Creative Team (2008). ActionScript 3.0 for Adobe Flash CS4 Professional Classroom in a Book. Berkeley, CA, USA: Peachpit Press.

Fowler, M. (1999). Refactoring Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley.

Fuhrer, R.M., Saraswat, V. (2009). Concurrency Refactoring for x10. Proceedings of the 3rd ACM Workshop on Refactoring Tools. ACM: New York, NY, USA.

Garrett, J. (2005). Ajax: A New Approach to Web Applications.
http://www.adaptivepath.com/ideas/essays/archives/000385.php.

Garrido, A., Johnson, R. (2003). Refactoring C with Conditional Compilation. Proceedings of the 18th IEEE International Conference on Automated Software Engineering. IEEE Computer Society: Washington, DC, USA, pp. 323-326.

Garrido, A., Rossi, G., Distante, D. (2011). Refactoring for Usability in Web Applications. IEEE Software 28(3), pp. 60-67.

Ghodrat, M.A., Givargis, T., Nicolau, A. (2007). Short-Circuit Compiler Transformation: Optimizing Conditional Blocks. Proceedings of the 12th Conference on Asia South Pacific Design Automation. IEEE Computer Society: Washington, DC, USA, pp. 504-510.

Ghosh, P., Rau-Chaplin, A. (2006). Performance of Dynamic Web Page Generation for Database-driven Web Sites. Proceedings of the International Conference on Next Generation Web Services Practices. IEEE Computer Society: Washington, DC, USA, pp. 56-63.

Goldstein, E.B. (2007), Cognitive Psychology: Connecting Mind, Research and Everyday Experience. Florence, KY: Cengage Learning.

Gough, B. (2004). An Introduction to GCC. Bristol, UK: Network Theory.

Grant, B., Philipose, M., Mock, M., Chambers, C., Eggers, S.J. (1999). An Evaluation of Staged Run-time Optimizations in DyC. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM: New York, NY, USA, pp. 293-304.

Griswold, W.G. (1991). Program Restructuring as an Aid to Software Maintenance. Ph.D. Thesis, University of Washington.

Grossman, G. (2006). ActionScript 3.0 and AVM2: Performance Tuning, http://www.onflex.org/ACDS/AS3TuningInsideAVM2JIT.pdf.

Hagen, W.V. (2006). The Definitive Guide to GCC. Berkeley, CA, USA: Apress.

Harold, E.R. (2008). Refactoring HTML: Improving the Design of Existing Web Applications. Upper Saddle River, NJ, USA: Addison-Wesley.

Hewlett-Packard Company. (2001). JavaTM Terminology HotSpot JVM Runtime Compiler. http://docs.hp.com/en/JAVAPERFTUTOR/02Terminology.pdf.

Hewlett-Packard Company. (2007). Load Testing 2.0 for Web 2.0: Simplifying Performance Validation for Rich Internet Applications [White paper]. http://www.webbuyersguide.com/resource/white-paper/9794/Load-Testing-20-for-Web-20-Simplifying-Performance-Validation-for-Rich-Internet-Applications.

Hiser, J.D., Kumar, N., Zhao, M., Zhou, S.K., Childers, B.R., Davidson, J.W., Soffa, M.L. (2006). Techniques and Tools for Dynamic Optimization. Proceedings of the 20th International Conference on Parallel and Distributed Processing. IEEE Computer Society: Washington, DC, USA.

Irani, R. (2010). JSON Continues its Winning Streak Over XML. http://blog.programmableweb.com/2010/12/03/json-continues-its-winning-streak-over-xml/.

Jacobs, S. (2006). Beginning XML with DOM and Ajax: from Novice to Professional. Berkeley, CA, USA: Apress.

Jain, P., Gupta, D. (2009). An Algorithm for Dynamic Load Balancing in Distributed Systems with Multiple Supporting Nodes by Exploiting the Interrupt Service. International Journal of Recent Trends in Engineering (IJRTE) 1(1), pp. 232-236.

Kanat-Alexander, M. (2008). Code Simplicity: Software Design in Open Source Projects. http://www.codesimplicity.com/wp-content/uploads/2008/07/code-simplicity-open-source-design.pdf.

Kaplan, I. (1999). Why Use ANTLR? http://www.bearcave.com/software/antlr/antlr_expr.html.

Kataoka, Y., Imai, T., Andou, H., Fukaya, T. (2002). A Quantitative Evaluation of Maintainability Enhancement by Refactoring. Proceedings of the International Conference on Software Maintenance. IEEE Computer Society: Washington, DC, USA, pp. 576-585.

Kazoun, C., Lott, J. (2008). Programming Flex 3. Sebastopol, CA, USA: O'Reilly Media.

Kennedy, K., McKinley, K.S., Tseng, C.W. (1991). Interactive Parallel Programming Using the Parascope Editor. IEEE Transactions on Parallel and Distributed Systems 2(3), pp. 329-341.

Kerievsky, J. (2004). Refactoring to Patterns. Boston, MA, USA: Addison-Wesley.

Khatchadourian, R., Muskalla, B. (2010). Enumeration Refactoring: A Tool for Automatically Converting Java Constants to Enumerated Types. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ACM: New York, NY, USA, pp.181-182.

Kiezun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.(2007). Refactoring for Parameterizing Java Classes. Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society: Washington, DC, USA, pp. 437-446.

Kim, C.G., Park, J.W., Lee, J.H., Kim, S.D. (2008). A Small Data Cache for Multimedia-oriented Embedded Systems. Journal of Systems Architecture 54(1-2), pp. 161-176.

King, A.B. (2008). Website Optimization: Speed, Search Engine & Conversion Rate Secrets. Sebastopol, CA, US: O'Reilly Media.

Kjolstad, F., Dig, D., Acevedo, G. Snir, M. (2009). Transformation for Class Immutability. Proceedings of the 33rd International Conference on Software Engineering. ACM: New York, NY, USA, pp. 61-70.

Lasky, J.A., Kevin, H. (1993), Conflict Resolution (CORE) for Software Quality Factors. Technical Report, Rochester Institute of Technology.

Lee, H., Dincklage, D.V., Diwan, A., Eliot, J., Moss, B. (2006). Understanding the Behavior of Compiler Optimizations. Software Practice & Experience 36(8), pp. 835-844.

Lee, P., Leone, M. (1996). Optimizing ML with Run-time Code Generation. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM: New York, NY, USA. 31(5), pp. 137-148.

Lekatsas, H., Wolf, W. (1999). SAMC: A Code Compression Algorithm for Embedded Processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(12), pp. 1689-1701.

Leone M., Dybvig, R.K. (1997). Dynamo: A Staged Compiler Architecture for Dynamic Program Optimization. Technical Report, Indiana University.

Li, Y.-T.S., Malik, S. (1997). Performance Analysis of Embedded Software Using Implicit Path Enumeration. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16(12), pp. 1477-1487.

Liao, S.-W., Diwan, A., Bosch, R.P. Jr., Ghuloum, A., Lam, G.M. (1999). SUIF Explorer: An Interactive and Interprocedural Parallelizer. Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM: New York, NY, USA, pp. 37-48.

Lindley, C.(2009). jQuery Cookbook. Sebastopol, CA: O'Reilly Media.

Lindvall, M., Tvedt, R.T., Costa, P. (2003). An Empirically-Based Process for Software Architecture Evaluation. Empirical Software Engineering 8(1), pp. 83-108.

Lott, J., Peters, K., Schall, D. (2008), ActionScript 3.0 Cookbook. Sebastopol, CA, USA: O'Reilly Media.

Luo, Q., Naughton, J.F., Xue, W. (2008). Form-based Proxy Caching for Database-backed Web Sites: Keywords and Functions. The Vldb Journal – VLDB 17(3), pp. 489-513.

Mancl, D.(2001). Refactoring for Software Migration. IEEE In Communications Magazine 39(10), pp. 88-93.

Matthews, J., Findler, R.B., Graunke, P.T., Krishnamurthi, S., Felleisen, M., (2001). Automatically Restructuring Programs for the Web. Proceedings of the 16th Annual International Conference on Automated Software Engineering. IEEE Computer Society: Washington, DC, USA pp. 211-222.

Mclellan, D. (2002). Flash Satay: Embedding Flash While Supporting Standards. http://www.alistapart.com/articles/flashsatay/.

Méndez, M., Overbey, J., Garrido, A., Tinetti, F., Johnson, R. (2010). A Catalog and Two Possible Classifications of Fortran Refactorings. Technical Report.

Mendonga, N.C., Maia, P.H.M., Fonseca, L.A., Andrade, R.M.C. (2004). RefaX: A Refactoring Framework Based on XML. In Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 147-156.

Mens, T., Tourwé, T. (2004). A Survey of Software Refactoring. IEEE Transactions on Software Engineering 30(2), pp. 126-139.

Mesbah, A., Deursen, A.V. (2007). Migrating Multipage Web Applications to Single-page AJAX Interfaces. Proceedings of the 11th European Conference on Software Maintenance and Reengineering. IEEE Computer Society: Washington, DC, USA, pp. 181-190.

Miller, G.A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review 63, pp. 81-97.

Mitchell, S. (2000). Designing Active Server Pages. Sebastopol, CA: O'Reilly Media.

Moock, C. (2007). Essential ActionScript 3.0. Sebastopol, CA, USA: O'Reilly Media.

Murphy, C., Persson, N. (2008). HTML and CSS Web Standards Solutions: A Web Standardistas' Approach. Berkeley, CA, USA: Friends of ED.

Murphy-Hill, E. (2007). Programmer-Friendly Refactoring Tools. Thesis Proposal, Portland State University.

Murphy-Hill, E., Parnin, C.P., Black, A.P. (2009). How We Refactor, and How We Know It. Proceedings of the IEEE 31st International Conference on Software Engineering. IEEE Computer Society: Washington: DC, USA, pp. 287-297.

Myers, D.S, Carlisle, J.N., Cowling, J.A, Liskov, B.H. (2007). MapJAX: Data Structure Abstractions for Asynchronous Web Applications. Proceedings of the USENIX Annual Technical Conference. USENIX Association: Berkeley, CA, USA.

Nelson, S. (2008). What's Wrong With 90% of Software Written Today? http://it.toolbox.com/blogs/tricks-of-the-trade/whats-wrong-with-90-of-software-written-today-21570.

Ng, W., Lam, W.Y., Cheng, J. (2006). Comparative Analysis of XML Compression Technologies. World Wide Web 9(1), pp. 5-33.

Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C. (2009). Comparison of JSON and XML Data Interchange Formats: A Case Study. Proceedings of the International Conference on Computer Applications in Industry and Engineering. ISCA: Cary, NC, USA, pp. 157-162.

Olsina, L., Rossi, G., Garrido, A., Distante, D., Canfora, G. (2007). Incremental Quality Improvement in Web Applications Using Web Model Refactoring. Proceedings of the International Conference on Web Information Systems Engineering, pp. 411-422.

Opdyke, W.F. (1992). Refactoring Object-Oriented Frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign.

Othman, O., Schmidt, D.C. (2001). Optimizing Distributed system Performance via Adaptive Middleware Load Balancing. Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems.

Pan, Z.L., Eigenmann, R. (2006). Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society: Washington, DC, USA.

Panda, P.R., Catthoor, F., Dutt, N.D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.G. (2001). Data and Memory Optimization Techniques for Embedded Systems. ACM Transactions on Design Automation of Electronic Systems (TODAES) 6(2), pp. 149-206.

Parr, T. (2007). The Definitive Antlr Reference: Building Domain-specific Languages. Raleigh, NC, USA: Pragmatic.

Paulson, L.D. (2005). Building Rich Web Applications with Ajax. Computer 38(10), pp. 14-17.

Polanco J. (2007). Flash Internals: Just-in-time (JIT) Compilation. http://blog.vivisectingmedia.com/2007/11/flash-internals-jit-and-garbage-collection-part-four/comment-page-1/#comment-928.

Ramaswamy, L., Liu, L., Arun, I. (2007). Scalable Delivery of Dynamic Content Using a Cooperative Edge Cache Grid. IEEE Transactions on Knowledge and Data Engineering 19(5), pp. 614-630.

Ricca, F., Tonella, P. (2001). Understanding and Restructuring Web Sites with ReWeb. IEEE MultiMedia 8(2), pp. 40-51.

Ricca, F., Tonella, P., Baxter, I.D. (2002). Web Application Transformations based on Rewrite Rules. Information and Software Technology 44(13), pp. 811-825.

Roberts, D.B. (1999). Practical Analysis for Refactoring. Technical Report, University of Illinois at Urbana-Champaign.

Roock, S., Havenstein, A. (2002). Refactoring Tags for Automatic Refactoring of Framework Dependent Applications. Proceedings of the Internetional Conference on Extreme Programming and Flexible Processes in Software Engineering XP. pp. 182-185.

Rossi, G., Urbieta, M., Ginzburg, J., Distante, D., Garrido, A. (2008). Refactoring to Rich Internet Applications. A Model-Driven Approach. In: 8th International Conference on Web Engineering. IEEE Computer Society: Los Alamitos, CA, USA, pp. 14-18.

Sanders, W. B., Cumaranatunge, C. (2007). ActionScript 3.0 Design Patterns. Sebastopol, CA, USA: O'Reilly Media.

Savoia, A. (2001).Web Page Response Time 101 (Understanding and Measuring Performance Test Results).
http://ericgoldsmith.com/__oneclick_uploads/2009/02/web_page_response_time_101.pdf.

Schäfer, M., Sridharan, M., Dolby, J., Tip, F. (2011). Refactoring Java Programs for Flexible Locking. Proceedings of the International Conference on Software Engineering. ACM, New York, NY, USA, pp. 71-80.

Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R. (2002). XMark: A Benchmarkfor XML Data Management. Proceedings of the Very Large Database Conference. VLDB Endowment. pp 974-985.

Schmitt, C. (2005). Professional CSS: Cascading Style Sheets for Web design. New York, NY, USA: John Wiley and Sons.

Seong, S.W., Mishra, P. (2006). A Bitmask-based Code Compression Technique for Embedded Systems. Proceedings of the IEEE/ACM International Conference on Computer-aided Design. ACM: New York, NY, USA. 27(4), pp. 673-685.

Shupe, R., Rosser, Z. (2007). Learning ActionScript 3.0. Sebastopol, CA, USA: O'Reilly Media.

Šimunić, T., Benini, L., Micheli, G.D., Hans, M. (2000). Source Code Optimization and Profiling of Energy Consumption in Embedded Systems Proceedings of the 13th International Symposium on System Synthesis. IEEE Computer Society: Washington, DC, USA, pp. 193-198.

Skinner G. (2007). Resource Management Strategies in Flash Player 9.
http://www.adobe.com/devnet/flashplayer/articles/resource_management.html.

Sluis, B.V.D. (2007). Flash Embedding Cage Match.
http://www.alistapart.com/articles/flashembedcagematch/.

Sluis, B.V.D. (2008). Flash Embed Test Suite.
http://www.bobbyvandersluis.com/flashembed/testsuite/.

Smashing Magazine.(2008). Web Form Design Patterns: Sign-Up Forms.
http://www.smashingmagazine.com/2008/07/04/web-form-design-patterns-sign-up-forms/.

Spinellis, D. (2006). Code Quality: The Open Source Perspective. Boston, MA, USA: Addison Wesley.

Srivastava A., Edwards, A., Vo, H. (2001). Vulcan: Binary Transformation in a Distributed Environment. Technical Report. Microsoft Research.

Srivisut, K., Muenchaisri, P. (2007). Defining and Detecting Bad Smells of Aspect-Oriented Software. Proceedings of the 31st Annual IEEE International Computer Software and Applications Conference, pp. 65-70.

Starr, J. (2008). Embed Flash or Die Trying. http://perishablepress.com/press/2007/04/17/embed-flash-or-die-trying.

Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.-M. (2001). Refactoring UML Models. Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer-Verlag London, UK.

Tahvildari, L., Kontogiannis, K. (2004). Improving Design Quality Using Meta-pattern Transformations: A Metric-based Approach. Journal of Software Maintenance and Evolution: Research and Practice 16(4-5), pp. 331-361.

Thompson, S., Reinke C.(2002). A Catalogue of Function Refactorings. Lab Report, University of Kent.

Tokuda, L., Batory, D. (2001). Evolving Object-oriented Designs With Refactorings. Proceedings of the 14th IEEE International Conference on Automated Software Engineering, pp. 174-181.

Vingralek, R., Breitbart, Y., Sayal, M., Scheuermann, P. (1999). Web++: A System for Fast and Reliable Web Service. Proceedings of the USENIX Annual Technical Conference. USENIX Association: Berkeley, CA, USA, pp. 171-184.

W3C. (2008). Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml/.

W3C. (2009). Namespaces in XML 1.0. http://www.w3.org/TR/REC-xml-names/#NT-LocalPart.

Weber, B., Reichert, M., Mendling, J., Reijers, H.A. (2011). Refactoring Large Process Model Repositories. Computers in Industry 62 (5), pp. 467-486.

Webucator. (2009). JavaScript Object Notation (JSON). http://www.learn-ajax-tutorial.com/Json.cfm.

Wloka, J., Sridharan, M. Tip, F. (2009). Refactoring for Reentrancy. Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM: New York, NY, USA, pp. 173-182.

Zakas, N.C., McPeak, J., Fawcett, J. (2007). Professional Ajax. Indianapolis, IN: Wiley.

Zmily, A., Killian, E., Kozyrakis, C. (2005). Improving Instruction Delivery with a Block-Aware ISA. Proceedings of the 11th International Euro-Par Conference. Springer: Germany, pp. 530-539.

Zmily, A., Kozyrakis, C. (2005). Energy-Efficient and High-Performance Instruction Fetch using a BlockAware ISA. Proceedings of the International Symposium on Low Power Electronics and Design. ACM: New York, NY, USA, pp. 36-41.

Zmily, A., Kozyrakis, C. (2006). Simultaneously Improving Code Size, Performance, and Energy in Embedded Processors. Proceedings of the Conference on Design, Automation and Test in Europe. European Design and Automation Association: Leuven, Belgium, pp. 224-229.

Zulzalil, H., Ghani, A.A.A., Selamat, M.H., Mahmod, R. (2008), A Case Study to Identify Quality Attributes Relationships for Web-based Applications. IJCSNS International Journal of Computer Science and Network Security 8(11), pp. 215-220.

Appendix A

In order to make it easier to read the grammar and to reduce the appearance of repeated grammar rules, several conventions are used.

1. To make the grammar simpler, only the directly utilized symbols and rules of the grammar are included.
2. Bolded rule name(s) within a rule indicates that the grammar continues to the corresponding rules in the subsequent statement(s).
3. If the grammar before refactoring has a "start_label -> end_label" line, it means this line can be replaced by the section of the grammar that begins with "start_label" and ends with "end_label" which has appeared prior to the "start_label -> end_label" line.
4. If the grammar after refactoring has a "start_label -> end_label" line, it means this line can be replaced by a section in the grammar before refactoring that begins with "start_label" and ends with "end_label".
5. A rule can be given an alias through the "=" symbol. For example, "ue1" is an alias of "unaryExpression" in the following statement, "ue1" can then be used later in the grammar after it is defined.

```
multiplicativeExpression: ue1=unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
```

Variables Refactoring Patterns

(1) Pattern name: Type annotations

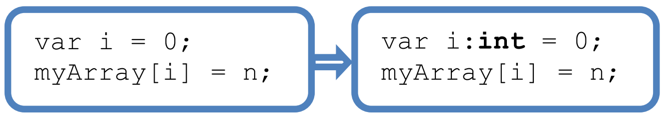
Problem: In ActionScript 2.0 (AS2), type annotations were just a coding aid, all values were dynamically typed atoms at runtime. However, in AS3, type information can be preserved till runtime (early binding). This improves performance and reduces memory consumption because it avoids unnecessary implicit type conversion. In addition, this also improves the system's type safety (Grossman, 2006). Type annotations are especially useful in Math operations and Object indexes.

Solution: When a variable is declared, always add type annotations.

Input: A type of a variable.

Recommend running environments: Same performance in all environments.

Example:



```
var i = 0;
myArray[i] = n;
```

```
var i:int = 0;
myArray[i] = n;
```

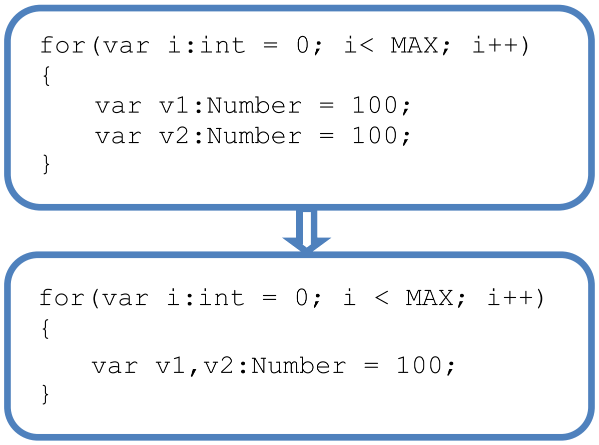
Grammar before refactoring:

```
variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration: variableIdentifierDecl (indexSuffix |
propertyReferenceSuffix | argumentSuffix)* (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)?;
```

Grammar after refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC? CONST? VAR? **variableDeclaration** (COMMA variableDeclaration)* semic;
variableDeclaration: **variableIdentifierDecl** (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? **COLON type;**

(2) Pattern name: Variables declaring fashion
Problem: Declaring variables in separate statements is slow because the “var” keyword is used many times.
Solution: Declare all the variables in a single statement.
Input: Permission to change.
Recommend running environments: Firefox 3.6 / Adobe Flash Player 9.
Example:



```
for(var i:int = 0; i< MAX; i++)
{
    var v1:Number = 100;
    var v2:Number = 100;
}
```

```
for(var i:int = 0; i < MAX; i++)
{
    var v1,v2:Number = 100;
}
```

Grammar before refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC? CONST? VAR? **variableDeclaration** semic;
variableDeclaration: **variableIdentifierDecl** (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? (COLON type)?;

Grammar after refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC? CONST? VAR? **variableDeclaration** (COMMA **variableDeclaration**)* semic;
variableDeclaration: **variableIdentifierDecl** (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? (COLON type)?;

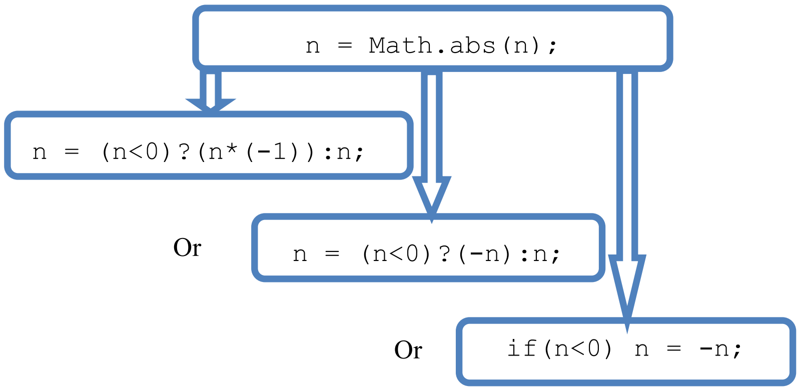
Objects Refactoring Patterns

Math and Operators Refactoring Patterns

(3)Pattern name: Math.abs
Problem: Math.abs is slow.
Solution: Replacements for Math.abs includes:

- Choice1: n<0?(n*(-1)):n
- Choice2: n<0?(-n):n
- Choice3: if(n<0)n=-n

Input: Selection of choices.
Recommend running environments for choice 1, 2 and 3: Same performance in all environments.
Example:



Grammar before refactoring:

```
expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression: unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . abs';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER;
```

Grammar after refactoring (Choice 1):

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression QUE
ae1=assignmentExpression COLON  ae2=assignmentExpression;
logicalORExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression LT se2=shiftExpression;
se1=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
se2=shiftExpression -> primaryExpression
primaryExpr: '0';
ae1=assignmentExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> additiveExpression
multiplicativeExpression:  ue1=unaryExpression STAR
ue2=unaryExpression;
ue1=unaryExpression -> primaryExpr
qualifiedName:  ARGUMENTIDENTIFIER;
ue2=unaryExpression ->
parExpression: LPAREN  expression  RPAREN;
expression -> shiftExpression
additiveExpression: me1=multiplicativeExpression SUB
me2=multiplicativeExpression;
me1=multiplicativeExpression -> primaryExpr
qualifiedName: VOID;
```

```
me2=multiplicativeExpression -> primaryExpression
primaryExpr: '-1';
ae2=assignmentExpression -> primaryExpr
qualifiedName:  ARGUMENTIDENTIFIER;
```

Grammar after refactoring (Choice 2):

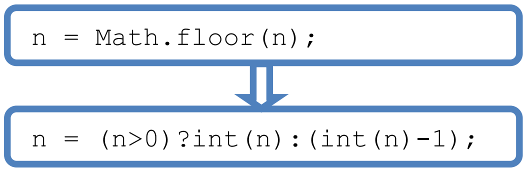
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression QUE
ae1=assignmentExpression COLON  ae2=assignmentExpression;
logicalORExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression LT se2=shiftExpression;
se1=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
se2=shiftExpression -> primaryExpression
primaryExpr: '0';
ae1=assignmentExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> shiftExpression
additiveExpression: me1=multiplicativeExpression SUB
me2=multiplicativeExpression;
me1=multiplicativeExpression -> primaryExpr
qualifiedName: VOID;
me2=multiplicativeExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
ae2=assignmentExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
```

Grammar after refactoring (Choice 3):

```
ifStatement:IF parExpression statement (ELSEIF parExpression
sstatement)?  (ELSE (WHITESPACE | EOL | COMMENT_MULTILINE |
COMMENT_SINGLELINE)* statement)?
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression LT se2=shiftExpression;
se1=additiveExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER
se2=additiveExpression -> primaryExpression
primaryExpr: '0';
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
```

```
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER;
assignmentExpression -> additiveExpression
me1=multiplicativeExpression SUB me2=multiplicativeExpression;
me1=multiplicativeExpression ->
qualifiedName: VOID;
me2=multiplicativeExpression ->
qualifiedName: ARGUMENTIDENTIFIER;
```

(4)Pattern name: Math.floor
Problem: Math.floor is slow.
Solution: Replace Math.floor(n) by (n>0)?int(n):(int(n)-1).
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:


```
n = Math.floor(n);
```
```
n = (n>0)?int(n):(int(n)-1);
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . floor';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER;
```

Grammar after refactoring:

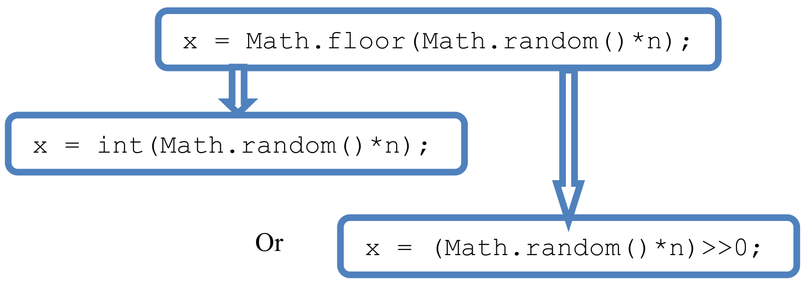
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression QUE
ae1=assignmentExpression COLON  ae2=assignmentExpression;
logicalORExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression GT se2=shiftExpression;
se1=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
se2=shiftExpression -> primaryExpression
primaryExpr: '0';
ae1=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'int';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
ae2=assignmentExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
```

```
parExpression: LPAREN  expression  RPAREN;
expression -> shiftExpression
additiveExpression: me1=multiplicativeExpression SUB
me2=multiplicativeExpression;
me1=multiplicativeExpression:  unaryExpression  ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'int';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
me2=multiplicativeExpression -> primaryExpression
primaryExpr: '1';
```

(5) Pattern name: Replace Math.floor(Math.random()*n) (n>0)
Problem: Math.floor(Math.random()*n) is slow.
Solution: Replacements for Math.floor(Math.random()*n) include:

- Choice1: int(Math.random()*n)
- Choice2:(Math.random()*n)>>0 (Choice 2 is faster than Choice 1)

Input: Selection of choices.
Recommend running environments for choice 1: Internet Explorer 8.0 / Adobe Flash Player 9 and Firefox 3.6 / Adobe Flash Player 9.
Recommend running environments for choice 2: Internet Explorer 8.0 / Adobe Flash Player 9 and 10, and Firefox 3.6 / Adobe Flash Player 10.
Example:


x = Math.floor(Math.random()*n);
x = int(Math.random()*n);
Or
x = (Math.random()*n)>>0;

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
```

logicalORExpression: logicalANDExpression (LOR logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS| ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression ((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU) additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB) multiplicativeExpression)*;
multiplicativeExpression: unaryExpression ((STAR|DIV|MOD)^ unaryExpression)*;
unaryExpression: unaryOp? **postfixExpression;**
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix | propertyReferenceSuffix | **argumentSuffix**)*;
memberExpression: primaryExpression | functionExpression | newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS | SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: 'Math . floor';
argumentSuffix: LPAREN **argumentList** RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> additiveExpression
multiplicativeExpression: ue1=**unaryExpression** STAR ue2=**unaryExpression;**
ue1=**unaryExpression:** unaryOp? **postfixExpression;**
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix | propertyReferenceSuffix | **argumentSuffix**)*;
memberExpression: primaryExpression | functionExpression | newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS | SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: 'Math . random';
argumentSuffix: LPAREN RPAREN;
ue2=**unaryExpression -> primaryExpr**
qualifiedName: IDENTIFIER;

Grammar after refactoring (Choice 1):

expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix | propertyReferenceSuffix | **argumentSuffix**)*;
memberExpression: primaryExpression | functionExpression |

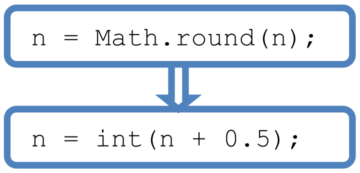
```
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'int';
argumentSuffix: LPAREN argumentList RPAREN;
assignmentExpression -> additiveExpression
multiplicativeExpression:  ue1=unaryExpression STAR
ue2=unaryExpression;
ue1=unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . random';
argumentSuffix: LPAREN RPAREN;
ue2=unaryExpression -> primaryExpr
qualifiedName: IDENTIFIER;
```

Grammar after refactoring (Choice 2):

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression -> relationalExpression
shiftExpression: ae1=additiveExpression SHR ae2=additiveExpression;
ae1=additiveExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> additiveExpression
multiplicativeExpression:  ue1=unaryExpression STAR
ue2=unaryExpression;
ue1=unaryExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Math . random';
argumentSuffix: LPAREN RPAREN;
ue2=unaryExpression -> primaryExpr
qualifiedName: IDENTIFIER;
ae2=additiveExpression -> primaryExpression
primaryExpr: '0';
```

(6) Pattern name: Math.round (n > 0)
Problem: Math.round is slow.
Solution: Replace Math.round(n) by int(n+0.5).
Input: Permission to change.

Recommend running environments: Same performance in all environments.
Example:



```
n = Math.round(n);
```

```
n = int(n + 0.5);
```

Grammar before refactoring:

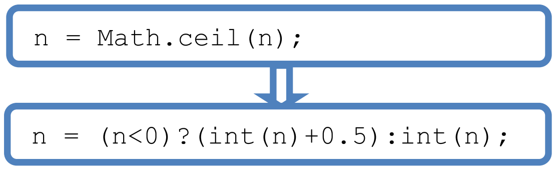
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . round';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
```

```
qualifiedName:  ARGUMENTIDENTIFIER;
```

Grammar after refactoring:

```
expression:  assignmentExpression ( COMMA  assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: 'int';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression-> shiftExpression
additiveExpression: me1=multiplicativeExpression PLUS
me1=multiplicativeExpression;
me1=multiplicativeExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
me2=multiplicativeExpression -> primaryExpression
primaryExpr: '0.5';
```

(7) Pattern name: Math.ceil
Problem: Math.ceil is slow.
Solution: Replace Math.ceil(n) by (n<0)?(int(n)+0.5):int(n).
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
n = Math.ceil(n);
```

```
n = (n<0)?(int(n)+0.5):int(n);
```

Grammar before refactoring:

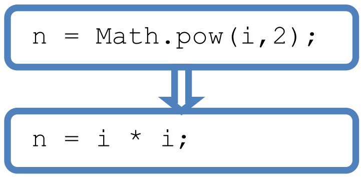
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
```

```
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . ceil';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER;
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression QUE
ae1=assignmentExpression COLON  ae2=assignmentExpression;
logicalORExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression LT se2=shiftExpression;
se1=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
se2=shiftExpression -> primaryExpression
primaryExpr: '0';
ae1=assignmentExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> shiftExpression
additiveExpression: me1=multiplicativeExpression PLUS
me2=multiplicativeExpression;
me1=multiplicativeExpression: unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix )*
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'int';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
me2=multiplicativeExpression -> primaryExpression
primaryExpr: '1';
ae2=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'int';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
```

(8) Pattern name: Math.pow
Problem: Math.pow is slow.
Solution: Replace Math.pow(i,2) by i*i.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
n = Math.pow(i,2);
```

```
n = i * i;
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
```

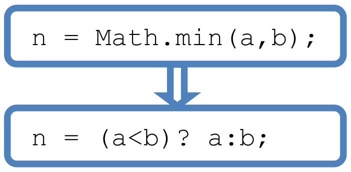
```
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . pow';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: ae1=assignmentExpression COMMA
as2=assignmentExpression;
ae1=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER;
ae2=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: '2';
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression -> additiveExpression
multiplicativeExpression:  unaryExpression STAR unaryExpression;
unaryExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER;
```

(9)Pattern name: Math.min
Problem: Math.min is slow.
Solution: Replace Math.min(a,b) by (a<b)?a:b.
Input: Permission to change.
Recommend running environments: Same performance in all environments.

Example:



```
n = Math.min(a,b);
```

```
n = (a<b)? a:b;
```

Grammar before refactoring:

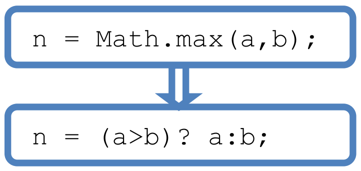
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . min';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: as1=assignmentExpression (COMMA
as2=assignmentExpression)*;
ae1=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
```

```
qualifiedName:  ARGUMENTIDENTIFIER1;
ae2=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER2;
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression QUE
ae1=assignmentExpression COLON  ae2=assignmentExpression;
logicalORExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression LT se2=shiftExpression;
se1=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER1;
se2=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER2;
ae1=assignmentExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER1;
ae2=assignmentExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER2;
```

(10) Pattern name: Math.max
Problem: Math.max is slow.
Solution: Replace Math.max(a,b) by (a>b)?a:b.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
n = Math.max(a,b);
```

```
n = (a>b)? a:b;
```

Grammar before refactoring:

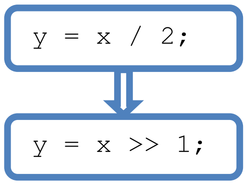
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . max';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: ae1=assignmentExpression COMMA
as2=assignmentExpression;
ae1=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER1;

ae2=assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  ARGUMENTIDENTIFIER2;
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
```

```
conditionalExpression: logicalORExpression QUE
ae1=assignmentExpression COLON  ae2=assignmentExpression;
logicalORExpression -> primaryExpression
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
parExpression: LPAREN  expression  RPAREN;
expression -> equalityExpression
relationalExpression: se1=shiftExpression GT se2=shiftExpression;
se1=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER1;
se2=shiftExpression -> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER2;
ae1=assignmentExpression-> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER1;
ae2=assignmentExpression-> primaryExpr
qualifiedName: ARGUMENTIDENTIFIER2;
```

(11) Pattern name: Replacing the division sign by the right-shift operator
Problem: The division sign is much slower than the right-shift operator.
Solution: If x is a signed integer, replace x/2 for x>>1.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9.
Example:


```
y = x / 2;
```
```
y = x >> 1;
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
```

```
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression: unaryExpression DIV unaryExpression;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: '2';
```

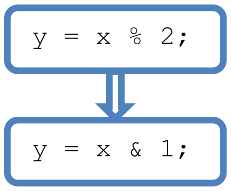
Grammar after refactoring:

```
expression -> relationalExpression
shiftExpression: additiveExpression SHR additiveExpression;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: '1';
```

(12) Pattern name: Replacing a modulo operator (%) by an AND (&) operator.
Problem: If the divisor is a power of 2, using the following formula to get modulus provides faster execution speed.

```
Modulus = Numerator & (Divisor - 1)
```

Solution: If the divisor is a power of 2, replace $x\%2^n$ by $x \& (2^n - 1)$ $(x>0)$.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
y = x % 2;
```

```
y = x & 1;
```

Grammar before refactoring:

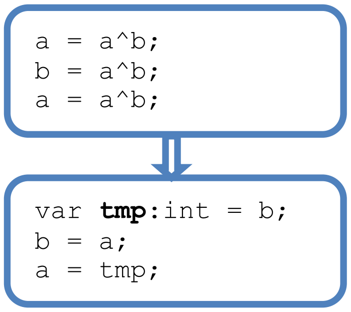
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
```

```
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression: unaryExpression MOD unaryExpression;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: '2';
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
bitwiseANDExpression: equalityExpression AND equalityExpression;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: '1';
```

(13) Pattern name: Avoid using Xor to swap variables
Problem: Without creating a new variable, Xor is used to swap variables. However, a more efficient method is to use a third variable to swap.
Solution: Create a new variable and use it to swap.
Input: The name for the new variable.
Recommend running environments: Same performance in all environments.
Example:



```
a = a^b;
b = a^b;
a = a^b;
```

```
var tmp:int = b;
b = a;
a = tmp;
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression XOR
bitwiseANDExpression;
bitwiseANDExpression: equalityExpression ( AND
equalityExpression)*
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
```

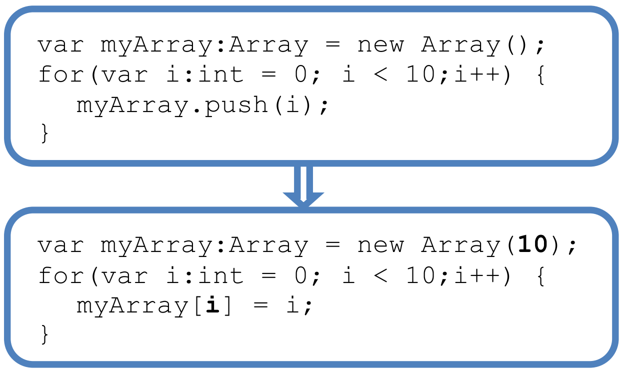
```
multiplicativeExpression: unaryExpression (STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpr:  (literal| arrayLiteral | objectLiteral | THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression) ;
qualifiedName:  IDENTIFIER;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
```

Arrays Refactoring Patterns

(14) Pattern name: Avoid using push() method to set a value in an array
Problem: The push() method is frequently used to set a value in an array in AS3. However, calling the push() method is costly; therefore, for arrays whose size are known, using an assignment statement as a substitution for the push() method provides increase in efficiency.
Solution: If the size of an array is known, indicate the size when declaring the array, and use the assignment operator instead of push() method to set an array value.
Input: The size of an array used for declaration and the index of an array used for value assignment.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9 and Firefox 3.6 / Adobe Flash Player 9.
Example:



```
var myArray:Array = new Array();
for(var i:int = 0; i < 10;i++) {
  myArray.push(i);
}
```

```
var myArray:Array = new Array(10);
for(var i:int = 0; i < 10;i++) {
  myArray[i] = i;
}
```

Grammar before refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
```

variableDeclaration: variableIdentifierDecl (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN **assignmentExpression**)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON **'Array'**;
assignmentExpression: conditionalExpression | leftHandSideExpression assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE assignmentExpression COLON assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression ((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU) additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB) multiplicativeExpression)*;
multiplicativeExpression: unaryExpression ((STAR|DIV|MOD)^ unaryExpression)*;
unaryExpression: unaryOp? **postfixExpression;**
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix | propertyReferenceSuffix | **argumentSuffix**)*;
memberExpression: primaryExpression | functionExpression | **newExpression;**
newExpression: NEW **memberExpression argumentSuffix***;
memberExpression: primaryExpression | functionExpression | newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS | SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: 'Array';
argumentSuffix: LPAREN RPAREN;
forStatement: FOR LPAREN (LineTerminator* forInit)? LineTerminator* SEMI expression? SEMI forUpdate? RPAREN **statement;**
statement: expression semic | **blockStatement** | useNamespaceStatement | namespaceStatement | constantVarStatement | tryStatement | labelledStatement | switchStatement | withStatement | returnStatement | breakStatement | continueStatement | forStatement | forInStatement | doWhileStatement | whileStatement | ifStatement | emptyStatement | variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement: LCURLY **statement*** RCURLY;
statement: expression semic | blockStatement | useNamespaceStatement | namespaceStatement | constantVarStatement | tryStatement | labelledStatement | switchStatement | withStatement | returnStatement | breakStatement | continueStatement | forStatement | forInStatement |

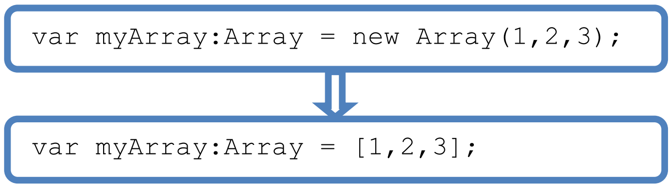
doWhileStatement | whileStatement | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | **argumentSuffix**)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS |
SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER **'.push'**;
argumentSuffix: LPAREN **argumentList** RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: IDENTIFIER;

Grammar after refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC?
CONST? VAR? **variableDeclaration** (COMMA variableDeclaration)* semic;
variableDeclaration: **variableIdentifierDecl** (indexSuffix |
propertyReferenceSuffix | argumentSuffix)* (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON **'Array'**;
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW **memberExpression argumentSuffix***;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS |
SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: **'Array'**;
argumentSuffix: LPAREN **argumentList** RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpression
primaryExpr: literal;
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN **statement;**
statement: expression semic | **blockStatement** |
useNamespaceStatement | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement: LCURLY **statement*** RCURLY;
statement: **expression** semic | blockStatement |
useNamespaceStatement | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement | ifStatement | emptyStatement

```
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER;
indexSuffix: LBRACK  expression  RBRACK;
expression -> primaryExpr
qualifiedName:  IDENTIFIER;
```

(15) Pattern name: Avoid using the new operator for objects
Problem: The new operator is typically used to instantiate objects and arrays; however, literal notation used to instantiate other data types (such as Number, String) can also be used on objects and arrays to speed up the applications.
Solution: If the size of an array is fixed, use the literal notation to instantiate the array instead of the new operator.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
var myArray:Array = new Array(1,2,3);
```

```
var myArray:Array = [1,2,3];
```

Grammar before refactoring:

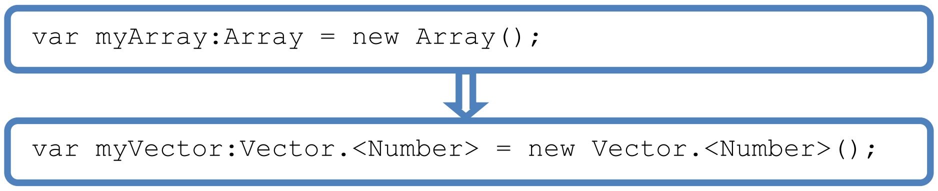
```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'Array';
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Array';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpression
primaryExpr: literal;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix     |     argumentSuffix)*     (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'Array';
assignmentExpression -> postfixExpression
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
arrayLiteral: LBRACK elementList RBRACK;
elementList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpression
primaryExpr: literal;
```

(16) Pattern name: Use Vector instead of Array (for Adobe Flash Player 10)
Problem: Vector, a new type of collection introduced by Adobe Flash Player 10, stores a collection of objects with specific type. Because it is a strongly typed container, Vectors allow Adobe Flash Player to deduce the type of objects inside the container. This results in faster code.

Solution: If the program will be executed in Adobe Flash Player 10, use Vector instead of Array.
Input: Permission to change the object type.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 10.
Example:



```
var myArray:Array = new Array();
```

```
var myVector:Vector.<Number> = new Vector.<Number>();
```

Grammar before refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC? CONST? VAR? **variableDeclaration** (COMMA variableDeclaration)* semic;

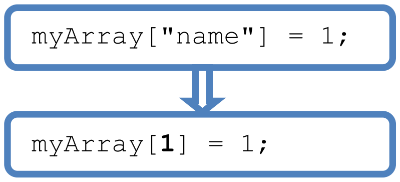
variableDeclaration: **variableIdentifierDecl** (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN **assignmentExpression**)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON **'Array'**;
assignmentExpression: conditionalExpression | leftHandSideExpression assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE assignmentExpression COLON assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression ((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU) additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB) multiplicativeExpression)*;
multiplicativeExpression: unaryExpression ((STAR|DIV|MOD)^ unaryExpression)*;
unaryExpression: unaryOp? **postfixExpression**;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix | propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression | **newExpression**;
newExpression: NEW **memberExpression argumentSuffix***;
memberExpression: primaryExpression | functionExpression |

```
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Array';
argumentSuffix: LPAREN argumentList? RPAREN;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'Vector';
propertyReferenceSuffix: DOT '<' IDENTIFIER '>';
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Vector';
propertyReferenceSuffix: DOT '<' IDENTIFIER '>';
argumentSuffix: LPAREN argumentList? RPAREN;
```

(17) Pattern name: Avoid using associative arrays
Problem: An associative array (sometimes called a hash or map) is an instance of the class Object, which uses named elements instead of numeric indexes. The named elements, named keys or properties, are a mapping from a string to the associated element value (Lott et al., 2008). However, accessing and setting values to an associative array takes longer time than accessing objects with numeric indexes.
Solution: Avoid using associative arrays if possible.
Input: The numeric array index.
Recommend running environments: Same performance in all environments.
Example:



```
myArray["name"] = 1;
```

```
myArray[1] = 1;
```

Grammar before refactoring:

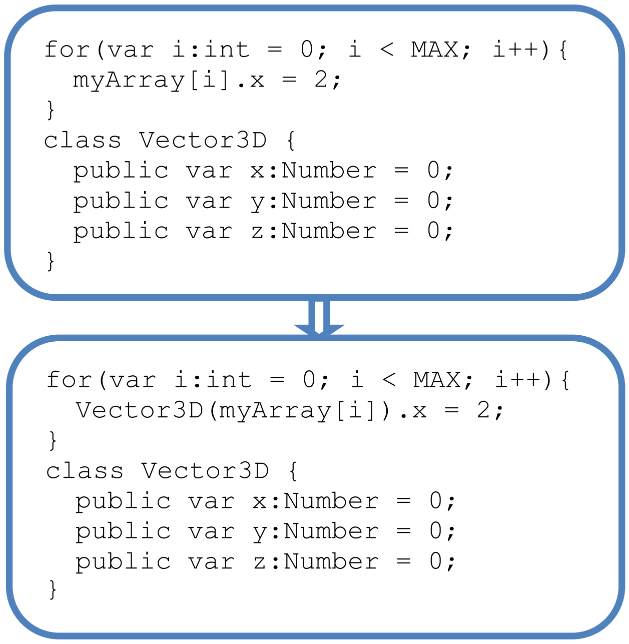
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
indexSuffix: LBRACK  QUO expression QUO RBRACK;
expression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: literal;
```

Grammar after refactoring:

```
expression -> callExpression
indexSuffix: LBRACK  expression RBRACK;
expression -> primaryExpr
primaryExpr: literal;
```

(18) Pattern name: Cast an object inside an array into a specific type
Problem: When accessing an array, making Adobe Flash Player know what type of data is inside can make array accessing faster.
Solution: Cast an object into a specific type when accessing an array.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 10.

Example: If the data type stored in the array "myArray" is an object Vector3D, then:



```
for(var i:int = 0; i < MAX; i++){
  myArray[i].x = 2;
}
class Vector3D {
  public var x:Number = 0;
  public var y:Number = 0;
  public var z:Number = 0;
}
```

```
for(var i:int = 0; i < MAX; i++){
  Vector3D(myArray[i]).x = 2;
}
class Vector3D {
  public var x:Number = 0;
  public var y:Number = 0;
  public var z:Number = 0;
}
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr:  (literal| arrayLiteral | objectLiteral | THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression) ;
qualifiedName:  IDENTIFIER;
indexSuffix: LBRACK  expression  RBRACK;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
```

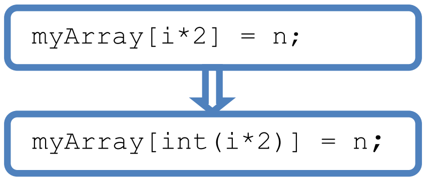
```
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr:  (literal| arrayLiteral | objectLiteral | THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression) ;
qualifiedName:  IDENTIFIER;
propertyReferenceSuffix: DOT  IDENTIFIER;
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr:  (literal| arrayLiteral | objectLiteral | THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  OBJECTIDENTIFIER;
argumentSuffix: LPAREN argumentList RPAREN;
argumentList -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression -> primaryExpr
qualifiedName:  IDENTIFIER;
indexSuffix: LBRACK  expression  RBRACK;
expression -> primaryExpr
qualifiedName:  IDENTIFIER;
propertyReferenceSuffix: DOT  IDENTIFIER;
```

(19) Pattern name: Avoid the promotion of numeric types
Problem: The semantics of ECMAScript requires the promotion of numeric types, so type int is often promoted to type Number. However, types int/uint provide faster array accessing than type Number.
Solution: When the array index contains calculations, explicitly casting to type int avoids the promotion from int to Number.
Input: Permission to change.
Recommend running environments: Same performance in all environments.

Example:



```
myArray[i*2] = n;
```

```
myArray[int(i*2)] = n;
```

Grammar before refactoring:

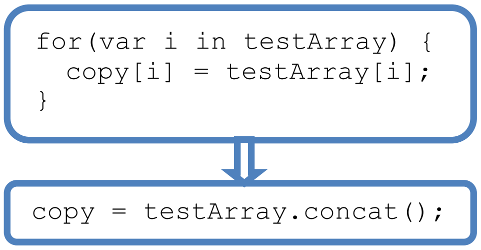
```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
indexSuffix: LBRACK  expression RBRACK;
expression: assignmentExpression (COMMA  assignmentExpression)*;
```

Grammar after refactoring:

```
expression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
indexSuffix: LBRACK  expression RBRACK;
expression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
```

```
qualifiedName: 'int';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
```

(20) Pattern name: Use array.concat() instead of for in loop to copy array members
Problem: For in loop is a common way to copy an array; however, array.concat() which concatenates the elements of an array provides the fastest way to copy an array.
Solution: Use concat() to copy an array.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
for(var i in testArray) {
  copy[i] = testArray[i];
}
```

```
copy = testArray.concat();
```

Grammar before refactoring:

```
forInStatement: FOR LPAREN forInControl RPAREN LCURLY? statement
RCURLY?;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:  LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
```

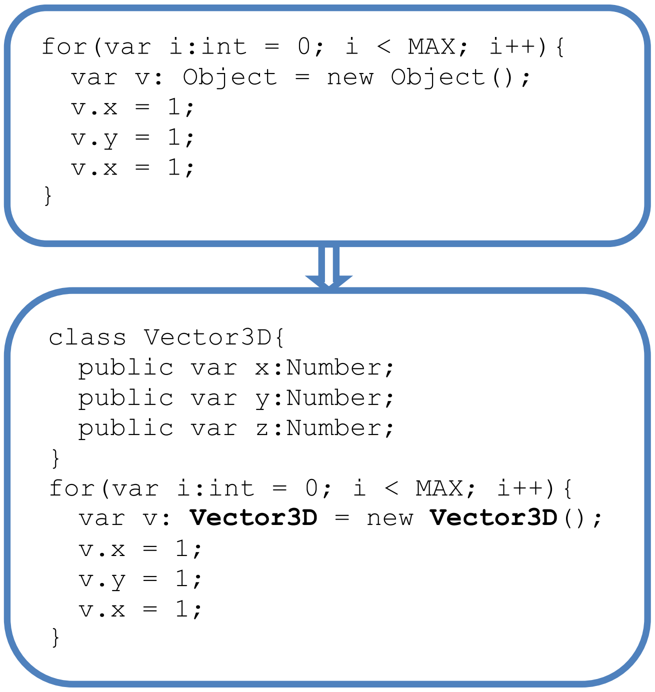
```
qualifiedName:  IDENTIFIER;
indexSuffix: LBRACK  expression  RBRACK;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  IDENTIFIER;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  IDENTIFIER;
indexSuffix: LBRACK  expression  RBRACK;
expression -> primaryExpr
qualifiedName:  IDENTIFIER;
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  IDENTIFIER;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  IDENTIFIER '. concat';
argumentSuffix: LPAREN RPAREN;
```

Other Objects Refactoring Patterns

(21) Pattern name: Use defined objects instead of the Object type
Problem: Avoid using the Object type. Only precise type annotations have the ability to improve running performance.
Solution: Use defined objects instead of the Object type.
Input: The class name and properties.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 10.
Example:



```
for(var i:int = 0; i < MAX; i++){
  var v: Object = new Object();
  v.x = 1;
  v.y = 1;
  v.x = 1;
}
```

```
class Vector3D{
  public var x:Number;
  public var y:Number;
  public var z:Number;
}
for(var i:int = 0; i < MAX; i++){
  var v: Vector3D = new Vector3D();
  v.x = 1;
  v.y = 1;
  v.x = 1;
}
```

Grammar before refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? (COLON
'Object');
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Object';
argumentSuffix: LPAREN argumentList? RPAREN;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? (COLON type);
assignmentExpression -> memberExpression
newExpression: NEW memberExpression argumentSuffix*;
primaryExpression: primaryExpr;
```

```
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: TYPEIDENTIFIER;
argumentSuffix: LPAREN argumentList? RPAREN;
```

(22) Pattern name: Do not use Regular Expression for searching

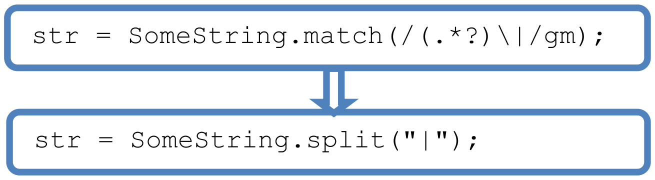
Problem: Regular Expression is great for validation, but not optimal for searching. When searching a string, use String methods because of their faster execution.

Solution: Use String methods for searching.

Input: Permission to change.

Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 10.

Example:



```
str = SomeString.match(/(.*?)\|/gm);
```

```
str = SomeString.split("|");
```

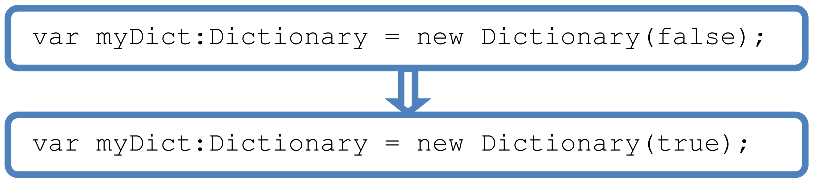
Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER '. match';
argumentSuffix: LPAREN argumentList RPAREN;
```

Grammar after refactoring:

```
Expression -> leftHandSideExpression
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER '.' STRINGMETHODIDENTIFIER;
argumentSuffix: LPAREN argumentList RPAREN;
```

(23) Pattern name: Add weak reference - Dictionary
Problem: The Garbage Collector is responsible for removing the objects that do not have any reference to any other active objects through reference counting or mark sweeping techniques. Weak references (Shupe & Rosser, 2007) are ignored by reference counting or marking. This means the object is eligible to be collected if all references to the object are weak references. In AS3, there is no way to remove an object from the memory. Thus, unless all the references to the object are deleted, the object will stay in the memory. Therefore, the weak references help to prevent memory leaks, so that the performance of the application will not be affected. There are only two places to add weak references in AS3. One place is with Dictionary objects. This pattern is presented to programmers to help increase the efficiency of their program. However, programmers should ensure that switching to weak references will not affect their program's correctness before using this refactoring pattern.
Solution: When declaring a Dictionary object, add weak references to that object.
Input: Permission to add weak references.
Example:



```
var myDict:Dictionary = new Dictionary(false);
```

```
var myDict:Dictionary = new Dictionary(true);
```

Grammar before refactoring:

```
variableStatement:   (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?   STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:    variableIdentifierDecl    (indexSuffix    |
propertyReferenceSuffix       |       argumentSuffix)*       (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
```

```
variableIdentifierDecl: IDENTIFIER COLON 'Dictionary';
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Dictionary';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> callExpression
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'false';
```

Grammar after refactoring:

```
variableStatement -> memberExpression
newExpression: NEW memberExpression argumentSuffix*;
memberExpression -> primaryExpr
qualifiedName: 'Dictionary';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList -> primaryExpr
qualifiedName: 'true';
```

Conditions Refactoring Patterns

(24) Pattern name: Rank if…else if… statements

Problem: The syntax for an if…else if… statement in AS3 is:

```
if(textExpression1){
   codeBlock1
}else if(textExpression2){
   codeBlock2
   …
}else{
   codeBlockN
}
```

If the branches are not executed in equal frequency, rank the branches from most frequently executed to least frequently executed.

Solution: Rewrite the if statements to place the branches in the order of most frequently executed to least frequently executed. The most frequently executed branch will be the top branch, and the least frequently executed branch will be the bottom branch.

Input: The ranking of the branches in an if statement.

Recommend running environments: Same performance in all environments.

Example:

```
if(x==0){
   trace("Error: The denominator is 0");
} else if(x<0){
   trace("Error: The result can not be negative");
} else {
   result = numerator / x;
}
```

⇓

```
if(x>0){
   result = numerator / x;
} else if(x<0){
   trace("Error: The result can not be negative");
} else {
   trace("Error: The denominator is 0");
}
```

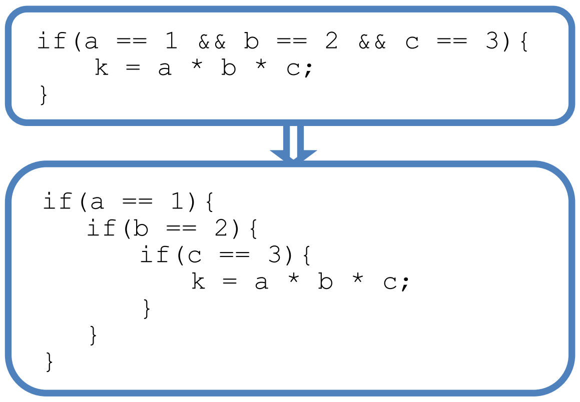
Grammar before refactoring:

```
ifStatement:IF parExpression stmt=statement  ELSE (WHITESPACE |
EOL | COMMENT_MULTILINE | COMMENT_SINGLELINE)* elsestmt=statement;
```

Grammar after refactoring:

```
ifStatement:IF parExpression elsestmt=statement  ELSE (WHITESPACE
| EOL | COMMENT_MULTILINE | COMMENT_SINGLELINE)* stmt=statement;
```

(25) Pattern name: Use nested if statements
Problem: && is regularly used when more than one condition is true in an if statement; however, nested if statement (using one if statement inside the other one) has higher efficiency than &&.
Solution: Use nested if statements instead of &&.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
if(a == 1 && b == 2 && c == 3){
    k = a * b * c;
}
```

```
if(a == 1){
   if(b == 2){
       if(c == 3){
           k = a * b * c;
       }
   }
}
```

Grammar before refactoring:

```
ifStatement:IF parExpression statement (ELSEIF parExpression
sstatement)?  (ELSE (WHITESPACE | EOL | COMMENT_MULTILINE |
COMMENT_SINGLELINE)* statement)?
parExpression: LPAREN  expression  RPAREN;
expression: assignmentExpression (COMMA  assignmentExpression)*;
```

Grammar after refactoring:

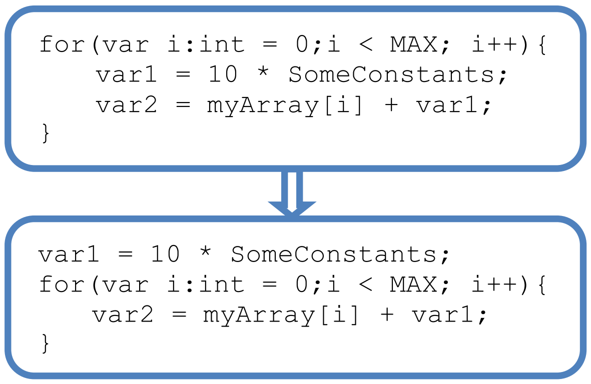
```
ifStatement:IF parExpression statement (ELSEIF parExpression
sstatement)?  (ELSE (WHITESPACE | EOL | COMMENT_MULTILINE |
COMMENT_SINGLELINE)* statement)?
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
```

Loops Refactoring Patterns

Pre-Operation Refactoring Patterns

(26) Pattern name: Pre-calculating the basic calculations
Problem: If some basic calculations are invariants inside a loop, move them before the loop to avoid the redundant calculations every time the loop executes.
Solution: Move the basic calculations out of the loop.

Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9 and Firefox 3.6 / Adobe Flash Player 9.
Example:



```
for(var i:int = 0;i < MAX; i++){
    var1 = 10 * SomeConstants;
    var2 = myArray[i] + var1;
}
```

```
var1 = 10 * SomeConstants;
for(var i:int = 0;i < MAX; i++){
    var2 = myArray[i] + var1;
}
```

Grammar before refactoring:

forStatement: FOR LPAREN (LineTerminator* forInit)? LineTerminator* SEMI expression? SEMI forUpdate? RPAREN **statement;**
statement: expression semic | **blockStatement** | useNamespaceStatement | namespaceStatement | constantVarStatement | tryStatement | labelledStatement | switchStatement | withStatement | returnStatement | breakStatement | continueStatement | forStatement | forInStatement | doWhileStatement | whileStatement | ifStatement | emptyStatement | variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement: LCURLY **statement*** RCURLY;
statement: **expression** semic | blockStatement | useNamespaceStatement | namespaceStatement | constantVarStatement | tryStatement | labelledStatement | switchStatement | withStatement | returnStatement | breakStatement | continueStatement | forStatement | forInStatement | doWhileStatement | whileStatement | ifStatement | emptyStatement | variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA assignmentExpression)*;

Grammar after refactoring:

expression: assignmentExpression (COMMA assignmentExpression)*;
forStatement: FOR LPAREN (LineTerminator* forInit)? LineTerminator* SEMI expression? SEMI forUpdate? RPAREN **statement;**

(27) Pattern name: Pre-calculating Math object's constants
Problem: Math object, a top-level object in AS3, has many public constants. Calling these constants is inefficient. Sometimes, these constants are used with some calculations, such as using ∏/180 to get radian. These calculations are quite slow, especially when used inside a loop. Therefore, pre-calculating these calculations (Math.PI/180) avoids the redundant calculations every time the loop executes.

Solution: (a) Create a new variable outside the loop; and (b) use this variable to execute the loop-invariant computations outside the loop.
Input: The name of the new variable.
Recommend running environments: Same performance in all environments.
Example: Math.PI/180 is widely used to draw graphics in ActionScript.

```
for(var i:int = 0; i < MAX; i++){
    radians[i] = degrees[i] * Math.PI/180;
}
```

```
var myPi:Number = Math.PI/180;
for(var i:int = 0; i < MAX; i++){
    radians[i] = degrees[i] * myPi;
}
```

Grammar before refactoring:

forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN **statement**;
statement: expression semic | **blockStatement** |
useNamespaceStatement | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement: LCURLY **statement*** RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression assignmentOperator **assignmentExpression**;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)

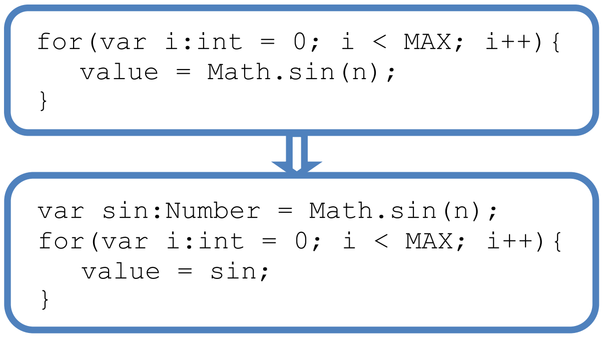
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB) multiplicativeExpression)*;
multiplicativeExpression: unaryExpression STAR ue1=**unaryExpression** DIV ue2=**unaryExpression;**
ue1=**unaryExpression:** unaryOp? **postfixExpression;**
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix | propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression | newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS | SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: 'Math.PI';
ue2=**unaryExpression -> primaryExpression**
primaryExpr: '180';

Grammar after refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC? CONST? VAR? **variableDeclaration** (COMMA variableDeclaration)* semic;
variableDeclaration: variableIdentifierDecl (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN **assignmentExpression**)? (AS IDENTIFIER)?;
variableIdentifierDecl: PIIDENTIFIER COLON **'Number';**
assignmentExpression -> additiveExpression
multiplicativeExpression: ue1=**unaryExpression** DIV ue2=**unaryExpression;**
ue1=**unaryExpression -> primaryExpr**
qualifiedName: 'Math.PI';
ue2=**unaryExpression -> primaryExpression**
primaryExpr: '180';
forStatement: FOR LPAREN (LineTerminator* forInit)? LineTerminator* SEMI expression? SEMI forUpdate? RPAREN **statement;**
statement: expression semic | **blockStatement** | useNamespaceStatement | namespaceStatement | constantVarStatement | tryStatement | labelledStatement | switchStatement | withStatement | returnStatement | breakStatement | continueStatement | forStatement | forInStatement | doWhileStatement | whileStatement | ifStatement | emptyStatement | variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement: LCURLY **statement*** RCURLY;
statement: expression semic | blockStatement | useNamespaceStatement | namespaceStatement | constantVarStatement | tryStatement | labelledStatement | switchStatement | withStatement | returnStatement | breakStatement | continueStatement | forStatement | forInStatement | doWhileStatement | whileStatement | ifStatement | emptyStatement | variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression: conditionalExpression | leftHandSideExpression assignmentOperator **assignmentExpression;**
assignmentExpression -> additiveExpression
multiplicativeExpression: unaryExpression STAR **unaryExpression;**
unaryExpression -> primaryExpr

```
qualifiedName:  PIIDENTIFIER;
```

(28) Pattern name: Pre-calculating trigonometric functions
Problem: If trigonometric functions (such as sin() and cos()) are invariants inside the loop, move them before the loop to avoid the redundant calculation every time the loop executes.
Solution: Move the trigonometric functions out of the loop.
Input: Permission to change.
Recommend running environments: Same performance in all environments.
Example:



```
for(var i:int = 0; i < MAX; i++){
   value = Math.sin(n);
}
```

```
var sin:Number = Math.sin(n);
for(var i:int = 0; i < MAX; i++){
   value = sin;
}
```

Grammar before refactoring:

```
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:   LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . sin' | 'Math . cos' |'Math . tan' |'Math .
cot';
argumentSuffix: LPAREN argumentList RPAREN;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: TRIGONOMETRICIDENTIFIER COLON 'Number';
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'Math . sin' | 'Math . cos' |'Math . tan' |'Math .
cot';
argumentSuffix: LPAREN argumentList RPAREN;
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
statement: expression semic | blockStatement |
useNamespaceStatement                     | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:  LCURLY statement* RCURLY;
```

```
statement: expression semic | blockStatement |
useNamespaceStatement                      | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
assignmentExpression -> primaryExpr
qualifiedName: TRIGONOMETRICIDENTIFIER;
```

(29) Pattern name: Pre-accessing constants from other classes

Problem: Calling constants from other classes is quite slow, especially when the call is inside a loop.

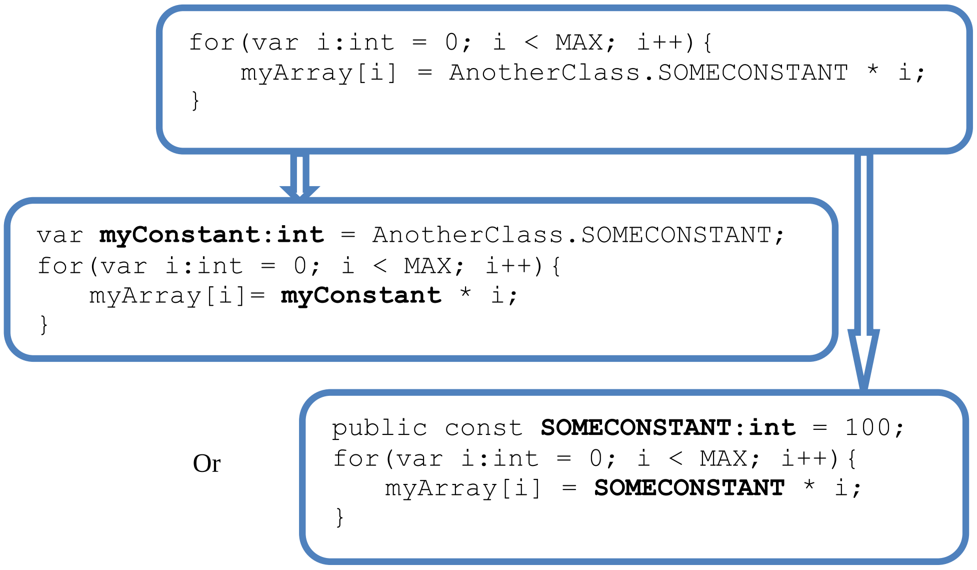
Solution:

- Choice 1: (a) Create a variable outside the loop; and (b) assign the value of the constant from another class to the new variable.
- Choice 2: (a) Create a local constant; and (b) assign the value of the constant from another class to the new constant.

Input: The name and type of the new constant.

Recommend running environments for choice 1 and 2: Same performance in all environments.

Example: If AnotherClass.SOMECONSTANT = 100:



```
for(var i:int = 0; i < MAX; i++){
    myArray[i] = AnotherClass.SOMECONSTANT * i;
}
```

```
var myConstant:int = AnotherClass.SOMECONSTANT;
for(var i:int = 0; i < MAX; i++){
    myArray[i]= myConstant * i;
}
```

Or

```
public const SOMECONSTANT:int = 100;
for(var i:int = 0; i < MAX; i++){
    myArray[i] = SOMECONSTANT * i;
}
```

Grammar before refactoring:

```
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
statement: expression semic | blockStatement |
useNamespaceStatement                      | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:  LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                  | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression STAR unaryExpression;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER (DOT IDENTIFIER)*;
```

Grammar after refactoring (Choice 1 and Choice 2):

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: CONSTANTIDENTIFIER COLON type;
```

```
assignmentExpression -> primaryExpr
qualifiedName: IDENTIFIER (DOT IDENTIFIER)*;
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
statement: expression semic | blockStatement |
useNamespaceStatement                        | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:   LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                        | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
assignmentExpression -> additiveExpression
multiplicativeExpression:  unaryExpression STAR unaryExpression;
unaryExpression -> primaryExpr
qualifiedName: CONSTANTIDENTIFIER;
```

(30) Pattern name: Pre-accessing methods from other classes

Problem: Calling methods from other classes is quite slow, especially when the call is inside a loop.

Solution: (a) Create a variable outside the loop; and (b) assign the method accessing to the new variable.

Input: The name and type of the new variable.

Recommend running environments for choice 1 and 2: Same performance in all environments.

Example:

```
for(var i:int = 0; i < MAX; i++){
    result = str.containsA("A");
}
```

⇓

```
var result1:Boolean = str.containsA("A");
for(var i:int = 0; i < MAX; i++){
    result = result1;
}
```

Grammar before refactoring:

```
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
```

```
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:  LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER (DOT IDENTIFIER)*;
argumentSuffix: LPAREN argumentList? RPAREN;
```

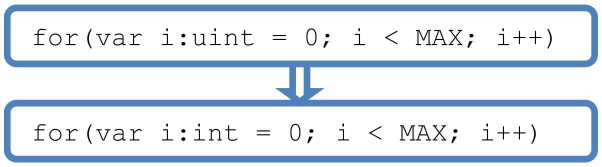
Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: VARIABLEIDENTIFIER COLON type;
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER (DOT IDENTIFIER)*;
argumentSuffix: LPAREN argumentList? RPAREN;
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:  LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
assignmentExpression -> primaryExpr
qualifiedName: VARIABLEIDENTIFIER;
```

For Loop Refactoring Patterns

(31) Pattern name: Replace type uint for iterations
Problem: UINT class is a new class introduced by AS3. It is similar to INT, except for the different range of values (0 to 4,294,967,295 (2^32-1)). UINT class is slow when it is used for iterations.
Solution: While initializing iterations, use type int instead of uint if the number of iteration is within the range of int.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9 and Firefox 3.6 / Adobe Flash Player 9.

Example:



```
for(var i:uint = 0; i < MAX; i++)
for(var i:int = 0; i < MAX; i++)
```

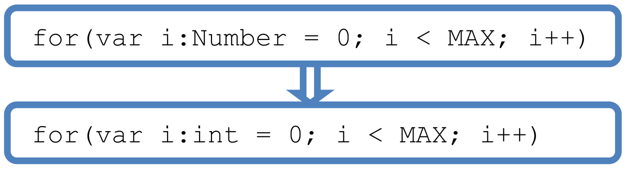
Grammar before refactoring:

forStatement: FOR LPAREN (LineTerminator* **forInit**)? LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
forInit: VAR **forvariableDeclarationList** | expression;
forvariableDeclarationList: **forvariableDeclaration**(COMMA forvariableDeclaration)*;
forvariableDeclaration: **forvariableIdentifierDecl** (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN assignmentExpression)? (AS IDENTIFIER)?;
forvariableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? (COLON **fortype**)? | THIS (DOT IDENTIFIER)? (COLON fortype)?;
fortype: **forqualifiedName** | STAR;
forqualifiedName: **'uint'**;

Grammar after refactoring:

forStatement -> fortype
forqualifiedName: **'int'**;

(32) Pattern name: Replace type Number for iterations
Problem: As mentioned in Array Refactoring Patterns, type int provides faster array accessing than type Number. Thus, when looping to access the members of an array, use type int instead of Number.
Solution: While initializing iterations, use type int instead of Number.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9.
Example:



```
for(var i:Number = 0; i < MAX; i++)
for(var i:int = 0; i < MAX; i++)
```

Grammar before refactoring:

forStatement: FOR LPAREN (LineTerminator* **forInit**)? LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
forInit: VAR **forvariableDeclarationList** | expression;
forvariableDeclarationList: **forvariableDeclaration**(COMMA forvariableDeclaration)*;

```
forvariableDeclaration: forvariableIdentifierDecl (indexSuffix |
propertyReferenceSuffix | argumentSuffix)* (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
forvariableIdentifierDecl: IDENTIFIER (DOT IDENTIFIER)? (COLON
fortype)? | THIS (DOT IDENTIFIER)? (COLON fortype)?;
fortype: forqualifiedName | STAR;
forqualifiedName: 'Number';
```

Grammar after refactoring:

```
forStatement -> fortype
forqualifiedName: 'int';
```

(33) Pattern name: Avoid array.length in for statements

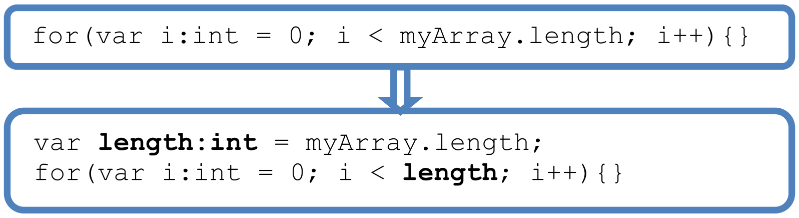
Problem: The AVM2 has the ability to perform Common Sub-expression Elimination (CSE) automatically; however, getter/setter sub-expressions are an exception which cannot be eliminated. Array.length is a getter/setter property, to optimize array.length, CSE by refactoring is necessary.

Solution: Create a new variable outside the loop; and use this variable to access array.length outside the loop.

Input: A name and type (int) for the length of the array.

Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9 and Firefox 3.6 / Adobe Flash Player 9.

Example:



```
for(var i:int = 0; i < myArray.length; i++){}
```

```
var length:int = myArray.length;
for(var i:int = 0; i < length; i++){}
```

Grammar before refactoring:

```
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression LT shiftExpression;
```

```
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression: unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER '.length';
```

Grammar after refactoring:

```
variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration: variableIdentifierDecl (indexSuffix |
propertyReferenceSuffix | argumentSuffix)* (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: LENGTHIDENTIFIER COLON 'int';
assignmentExpression -> primaryExpr
qualifiedName: IDENTIFIER '.length';
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
expression: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> equalityExpression
relationalExpression: shiftExpression LT shiftExpression;
shiftExpression -> primaryExpr
qualifiedName: LENGTHIDENTIFIER;
```

Other Loop Refactoring Patterns

(34) Pattern name: Avoid recreating an object to initialize the object
Problem: After an instance is created, don't recreate it when initializing the instance. The creation of unnecessary instances makes Adobe Flash Player slow, because extra time will be spent on creating the instance and on being collected by the Garbage Collector once the objects are no longer required.
Solution: Assign values to an object instead of recreating the object.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 9 and Firefox 3.6 / Adobe Flash Player 9.

Example:

```
for(var i:int = 0; i < MAX; i++){
    var myPoint:Point = new Point();
    myPoint = new Point(i, i + 2);
}
```

```
for(var i:int = 0; i < MAX; i++){
    var myPoint:Point = new Point();
    myPoint.x = i;
    myPoint.y = i + 2;
}
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  IDENTIFIER;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression ( AND
equalityExpression)*
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression: unaryExpression (STAR|DIV|MOD)^
unaryExpression)*
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
```

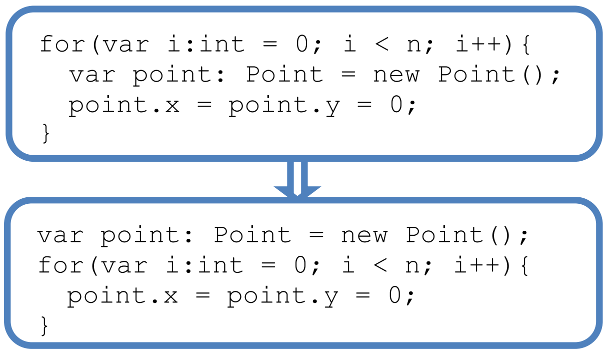
```
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
argumentSuffix: LPAREN argumentList? RPAREN;
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  IDENTIFIER (DOT IDENTIFIER)*;
assignmentExpression -> primaryExpr
qualifiedName:  IDENTIFIER;
```

(35) Pattern name: Avoid creating instances inside a loop
Problem: Calling the new operator is very expensive. To avoid recreating the instance every time the loop iterates, create an instance of a class outside the loop.
Solution: Move instance creations outside the loop.
Input: Permission to change.
Recommend running environments: Same performance in all environments.

Example:



```
for(var i:int = 0; i < n; i++){
  var point: Point = new Point();
  point.x = point.y = 0;
}
```

```
var point: Point = new Point();
for(var i:int = 0; i < n; i++){
  point.x = point.y = 0;
}
```

Grammar before refactoring:

```
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
blockStatement:   LCURLY statement* RCURLY;
statement: expression semic | blockStatement |
useNamespaceStatement                    | namespaceStatement |
constantVarStatement | tryStatement | labelledStatement |
switchStatement | withStatement | returnStatement | breakStatement
| continueStatement | forStatement | forInStatement |
doWhileStatement | whileStatement  | ifStatement | emptyStatement
| variableStatement | functionDeclaration | expressionNoIn semic;
variableStatement:   (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?   STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
```

Grammar after refactoring:

```
variableStatement:   (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?   STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
forStatement: FOR LPAREN (LineTerminator* forInit)?
LineTerminator* SEMI expression? SEMI forUpdate? RPAREN statement;
```

Packages, Classes and Functions Refactoring Patterns

(36) Pattern name: Import package.Class instead of package.*
Problem: When importing a package, only import the package.Class rather than package.*. If importing package.*, all the classes inside the package will be imported which are the “extra baggage” (Sanders & Cumaranatunge, 2007).
Solution: Remove package.*, use specified package.Class instead.
Input: Permission to change.

Example:

```
import flash.display.*;
import flash.text.*;
```

```
import flash.display.MovieClip;
import flash.display.Sprite;
import flash.text.TextField;
```

Grammar:

importDeclaration: IMPORT **importDeclarationTypeElement** semic;
importDeclarationTypeElement: IDENTIFIER
importDeclarationTypeElementPart*;
importDeclarationTypeElementPart: DOT (IDENTIFIER | STAR);

(37) Pattern name: Avoid using Dynamic classes
Problem: In AS3, a class can be dynamic which is allowed to add properties and methods at runtime, or sealed (by default) which cannot be altered at runtime (Florio, 2008). However, dynamic classes consume more memory to create internal hash tables to store dynamic properties and methods.
Solution: If dynamic classes are not necessary, they should be changed to sealed classes.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 10.
Example:

```
public dynamic class MyClass{}
//Add dynamic properties and methods
```

```
public class MyClass{
//Move dynamic properties and methods here
}
```

Grammar before refactoring:

classDeclaration: **memberModifiers**? CLASS type (EXTENDS type)? (IMPLEMENTS typeList)? classBody;
memberModifiers: (DYNAMIC | FINAL | INTERNAL | NATIVE | OVERRIDE | PRIVATE | PROTECTED | PUBLIC | STATIC | IDENTIFIER)+;

Grammar after refactoring:

classDeclaration: **memberModifiers**? CLASS type (EXTENDS type)? (IMPLEMENTS typeList)? classBody;
memberModifiers: (FINAL | INTERNAL | NATIVE | OVERRIDE | PRIVATE | PROTECTED | PUBLIC | STATIC | IDENTIFIER)+;

(38) Pattern name: Use static methods if possible
Problem: Static methods are faster than instance methods because no time is spent on the creation of an instance of a class when static methods, such as the methods in the Math class, are called.
Solution: Change an instance method to a static method.
Input: Permission to change.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 10.
Example:

```
private function myFunc():void{};
```

```
private static function myFunc():void{};
```

Grammar before refactoring:

```
functionDeclaration: functionDefination blockStatement;
functionDefination:  memberModifiers? FUNCTION
functionGetSetModifier? IDENTIFIER formalParameterList
functionReturnType?;
memberModifiers: (DYNAMIC | FINAL | INTERNAL | NATIVE | OVERRIDE |
PRIVATE | PROTECTED | PUBLIC | IDENTIFIER)+;
```

Grammar after refactoring:

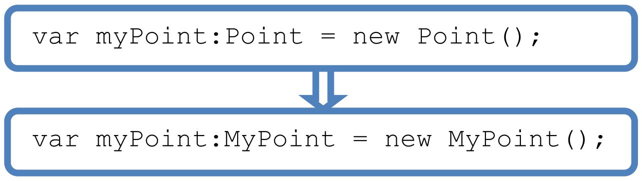
```
functionDeclaration: functionDefination blockStatement;
functionDefination:  memberModifiers? FUNCTION
functionGetSetModifier? IDENTIFIER formalParameterList
functionReturnType?;
memberModifiers: (DYNAMIC | FINAL | INTERNAL | NATIVE | OVERRIDE |
PRIVATE | PROTECTED | PUBLIC | STATIC | IDENTIFIER)+;
```

Graphic Display Refactoring Patterns

(39) Pattern name: Avoid using the flash.geom.point class
Problem: The Point class allows you to draw points using two properties: x and y. Though it is frequently used in ActionScript, this class significantly slows down Adobe Flash Player. Consequently, using a custom point class as an alternative to the flash.geom.point class increases the performance of Flash applications. For example, Nodename has designed a new MyPoint[97] class as a substitute for the Point class.
Solution: Using a custom point class instead of the flash.geom.point class.
Input: Permission to change.
Recommend running environments: Firefox 3.6 / Adobe Flash Player 9 and 10.

[97] http://nodename.com/blog/2005/09/26/point-class-slow/

Example:



```
var myPoint:Point = new Point();
```

```
var myPoint:MyPoint = new MyPoint();
```

Grammar before refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl    (indexSuffix    |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'Point';
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Point';
argumentSuffix: LPAREN argumentList? RPAREN;
```

Grammar after refactoring:

variableStatement: (PROTECTED|PRIVATE|PUBLIC|INTERNAL)? STATIC? CONST? VAR? **variableDeclaration** (COMMA variableDeclaration)* semic;
variableDeclaration: variableIdentifierDecl (indexSuffix | propertyReferenceSuffix | argumentSuffix)* (ASSIGN **assignmentExpression**)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON **'MyPoint'**;
assignmentExpression -> memberExpression
newExpression: NEW **memberExpression argumentSuffix***;
memberExpression: primaryExpression | functionExpression | newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral | THIS | SUPER | **qualifiedName** | xmlPrimaryExpression | parExpression);
qualifiedName: 'MyPoint';
argumentSuffix: LPAREN **argumentList?** RPAREN;

(40) Pattern name: Use Sprite objects instead of MovieClip objects
Problem: Sprite is a new class introduced by AS3. Sprite is similar to MovieClip, they both inherit from DisplayObject. However, MovieClip has a timeline, which has a significant overhead.
Solution: If the timeline is not necessary, MovieClip objects should be changed to Sprite objects.
Input: Permission to change object type.
Example:

```
import flash.display.MovieClip;

var myMovieClip:MovieClip = new MovieClip();
myMovieClip.graphics.beginFill(0xff0000);
myMovieClip.graphics.drawCircle(40, 40, 40);
myMovieClip.addEventListener(MouseEvent.CLICK, clicked);

function clicked(event:MouseEvent):void {
  trace("Click MovieClip!");
}
addChild(myMovieClip);
```

⇓

```
import flash.display.Sprite;

var mySprite:Sprite = new Sprite();
mySprite.graphics.beginFill(0xff0000);
mySprite.graphics.drawCircle(40, 40, 40);
mySprite.addEventListener(MouseEvent.CLICK, clicked);

function clicked(event:MouseEvent):void {
  trace("Click Sprite!");
}
addChild(mySprite);
```

Grammar before refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix     |     argumentSuffix)*     (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'MovieClip';
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'MovieClip';
argumentSuffix: LPAREN argumentList? RPAREN;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix     |     argumentSuffix)*     (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'Sprite';
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Sprite';
argumentSuffix: LPAREN argumentList? RPAREN;
```

(41) Pattern name: Speeding up access to getter properties
Problem: In AS3, a class is allowed to define setter methods to set properties and getter methods to access properties. Storing getter properties in a variable and accessing that variable is faster than using the getter properties directly.
Solution: Store getter properties in a local variable.
Input: The name of the new variable.
Recommend running environments: Internet Explorer 8.0 / Adobe Flash Player 10 and Firefox 3.6 / Adobe Flash Player 9 and 10.
Example:

```
var sprite:Sprite = new Sprite();
sprite.graphics.clear();
sprite.graphics.beginFill(0x000000);
sprite.graphics.drawCircle(0,10,0);
sprite.graphics.endFill();
```

```
var g:Graphics = sprite.graphics;
g.clear();
g.beginFill(0x000000);
g.drawCircle(0,10,10);
g.endFill();
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER '. graphics' (DOT IDENTIFIER)*;
argumentSuffix: LPAREN argumentList? RPAREN;
```

Grammar after refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix     |     argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: GRAPHICIDENTIFIER COLON 'Graphics';
assignmentExpression -> primaryExpr
qualifiedName: IDENTIFIER '. graphics';
expression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: GRAPHICIDENTIFIER (DOT IDENTIFIER)*;
argumentSuffix: LPAREN argumentList? RPAREN;
```

Event/Event Handling Refactoring Patterns

(42) Pattern name: Use Enter.Enter_FRAME instead of Timer
Problem: Both Enter.Enter_FRAME and Timer can be used to create animations. The differences between the two are (Moock, 2007):

(1) Enter.Enter_FRAME triggers on every frame, therefore, the time intervals are the same as the frame rate; a Timer dispatches TimerEvent.TIMER events at programmer-specified time intervals, not the frame rate;

(2) For the same animation, the code for Enter.Enter_FRAME is simpler than Timer;
(3) Timer requires more memory than Enter.Enter_FRAME because of its creation and event dispatch; and
(4) The updateAfterEvent() is a method in TimerEvent which is used to refresh the screen (to create a smooth animation). It forces Adobe Flash Player to render immediately after an event is processed. However, if each object has a separate Timer and each TimerEvent.TIMER event uses the updateAfterEvent() method, numerous independent requests to refresh the screen can cause performance problems.

Solution: If no change of frame rate is required, use Enter.Enter_FRAME instead of Timer.
Input: Permission to change from Timer to Enter.Enter_FRAME.
Recommend running environments: Same performance in all environments.
Example:

```
var myTimer:Timer = new Timer(delay, repeatCount);
myTimer.addEventListener(TimerEvent.TIMER, onTimerTick);
myTimer.start();

public function onTimerTick(event:TimerEvent):void{
   trace("TimerHandler:" + event);
}
```

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);

public function onEnterFrame(event:Event):void{
   trace("EnterFrame:" + event);
}
```

Grammar before refactoring:

```
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix     |     argumentSuffix)*     (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: IDENTIFIER COLON 'Timer';
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
```

```
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN) relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF) shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
newExpression: NEW memberExpression argumentSuffix*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: 'Timer';
argumentSuffix: LPAREN argumentList RPAREN;
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER '. addEventListener';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: 'TimerEvent.TIMER';
expression -> leftHandSideExpression
callExpression: memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName: IDENTIFIER '. start';
argumentSuffix: LPAREN RPAREN;
functionDeclaration: functionDefination blockStatement;
functionDefination:  memberModifiers? FUNCTION
functionGetSetModifier? IDENTIFIER formalParameterList
functionReturnType?;
formalParameterList: LPAREN (formalNonEllipsisParameter (COMMA
formalEllipsisParameter)?)? RPAREN;
formalNonEllipsisParameter: variableDeclaration (COMMA
variableDeclaration)*;
variableStatement:  (PROTECTED|PRIVATE|PUBLIC|INTERNAL)?  STATIC?
CONST? VAR? variableDeclaration (COMMA variableDeclaration)* semic;
variableDeclaration:   variableIdentifierDecl   (indexSuffix   |
propertyReferenceSuffix      |      argumentSuffix)*      (ASSIGN
assignmentExpression)? (AS IDENTIFIER)?;
variableIdentifierDecl: 'event' COLON 'TimerEvent';
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: leftHandSideExpression;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'addEventListener';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression ->  primaryExpr
qualifiedName: 'Event.ENTER_FRAME';
functionDeclaration -> variableDeclaration
variableIdentifierDecl: 'event' COLON 'Event';
```

(43) Pattern name: Add weak reference - addEventListener
Problem: The second place to add weak references is the .addEventListener() method in the EventDispatcher class. The .addEventListener() method registers an event to an object, and the object starts to listen to that event. If the listener is not used any more, it is good practice to remove it explicitly by using the .removeEventListener() method, otherwise the object cannot be collected by Garbage Collector and will stay in the memory until all the listeners are removed. However, “weak references allow the object to be deleted even if the event listener has not been explicitly removed” (Braunstein, 2007). Therefore, it should always be used to prevent memory leaks. There are five properties in the .addEventListener() method, the last property specifies the weak references (the default value is false). This pattern is presented to programmers to help increase the efficiency of their program. However, programmers should ensure that switching to weak references will not affect their program's correctness before using this refactoring pattern.
Solution: Use weak references when registering an event for an object.
Input: Permission to change.
Example:

```
addEventListener(MouseEvent.CLICK,clickHandler,false,0,false);
```

```
addEventListener(MouseEvent.CLICK,clickHandler,false,0,true);
```

Grammar before refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression: conditionalExpression |
leftHandSideExpression  assignmentOperator assignmentExpression;
conditionalExpression: logicalORExpression (QUE
assignmentExpression COLON  assignmentExpression)?;
logicalORExpression: logicalANDExpression (LOR
logicalANDExpression)*;
logicalANDExpression: bitwiseORExpression (LAND
bitwiseORExpression)*;
bitwiseORExpression: bitwiseXORExpression (OR
bitwiseXORExpression)*;
bitwiseXORExpression: bitwiseANDExpression (XOR
bitwiseANDExpression)*;
bitwiseANDExpression: equalityExpression (AND equalityExpression)*;
equalityExpression: relationalExpression ((EQ|NEQ|SAME|NSAME|IS|
ADD_ASSIGN|MUL_ASSIGN)  relationalExpression)*;
relationalExpression: shiftExpression
((IN|LT|GT|LTE|GTE|INSTANCEOF)  shiftExpression)*;
shiftExpression: additiveExpression ((SHL|SHR|SHU)
additiveExpression)*;
additiveExpression: multiplicativeExpression ((PLUS|SUB)
multiplicativeExpression)*;
multiplicativeExpression:  unaryExpression ((STAR|DIV|MOD)^
unaryExpression)*;
unaryExpression: unaryOp? postfixExpression;
postfixExpression: leftHandSideExpression postfixOp?;
leftHandSideExpression: callExpression | newExpression;
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'addEventListener';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: 'false';
```

Grammar after refactoring:

```
expression: assignmentExpression (COMMA  assignmentExpression)*;
assignmentExpression -> leftHandSideExpression
callExpression:  memberExpression (indexSuffix |
propertyReferenceSuffix | argumentSuffix)*;
memberExpression: primaryExpression | functionExpression |
newExpression;
primaryExpression: primaryExpr;
primaryExpr: (literal| arrayLiteral | objectLiteral |  THIS |
SUPER | qualifiedName | xmlPrimaryExpression | parExpression);
qualifiedName:  'addEventListener';
argumentSuffix: LPAREN argumentList RPAREN;
argumentList: assignmentExpression (COMMA assignmentExpression)*;
assignmentExpression -> primaryExpr
qualifiedName: 'true';
```