A
THESIS
ON

REFACTORING TECHNIQUES FOR JAVASCRIPT

Submitted in partial fulfillment of
the requirement for the award
of the degree of

MASTER OF TECHNOLOGY
(SOFTWARE ENGINEERING)

Submitted by:
ABHAY JOSHI
ROLL NO: 02/SE/09
REGISTRATION NO: 07/MT/SE/FT

Under the guidance of
DR. DAYA GUPTA
PROFESSOR, HEAD OF DEPARTMENT,
DEPARTMENT OF COMPUTER ENGINEERING,
DELHI TECHNOLOGICAL UNIVERSITY
BAWANA ROAD, DELHI-110042

DEPARTMENT OF COMPUTER ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
2009-2011

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY
BAWANA ROAD, DELHI-110042

Date: ____________

This is to certify that thesis entitled ***Refactoring Techniques for Javascript*** which is submitted by **Abhay Joshi** in partial fulfillment of the requirement for the award of **M.Tech.** degree in **Software Engineering** to **Delhi Technological University, Delhi** is a record of the candidate own work carried out by him under my supervision. The matter embodied in this thesis is original and has not been submitted for the award of any other degree.

DR. DAYA GUPTA
PROFESSOR, HEAD OF DEPARTMENT & PROJECT GUIDE
DEPARTMENT OF COMPUTER ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
BAWANA ROAD, DELHI-110042

ACKNOWLEDGEMENTS

I thank my project guide, **Dr. Daya Gupta, Professor and Head of Department, Department of Computer Science Engineering, Delhi Technological University,** for her support, guidance and patience during my studies at the Delhi Technological University. She gave me the freedom to explore the domain of refactoring. She invested her most valuable resource on my behalf: her time. She helped in pointing out places in several drafts of the thesis where clarity could be improved and claims made more precise.

I will also like to thank all the faculty members and staff members of Department of Computer Engineering at Delhi Technological University for sharing their knowledge and experiences with me as well as for their support.

I will like to thank **Prof. P.B. Sharma, Vice Chancellor, Delhi Technological University** for taking the initiative of starting the course of Master of Technology in Software Engineering at Delhi Technological University in the year 2009.

I will like to thank my batchmates at Delhi Technological University for sharing their ideas and opinions on several topics that were important for my work.

I will like to thank my parents and my elder brother for their patience and support. Without them, I never would have made through the program.

I owe thanks to **Martin Fowler** and many other people who shared information about refactoring on the Internet to make human knowledge free for all.

I owe thanks to all the volunteers in the open source movement that has provided high quality tools like Apache Web Server, PHP etc to students for transforming our ideas to reality. They have done a commendable job by helping people in getting rid of proprietary softwares.

Finally, I am thankful to God for having granted me the skills and opportunities that made this work possible.

ABSTRACT

In the past decade a lot of IDEs (Integrated Development Environment) like Visual Studio[53], Eclipse[54], Netbeans[55] etc. have embedded refactoring support in their software packages. Yet, they are merely supporting a few specific refactoring techniques (like move method, extract method and rename method). Moreover many of these IDEs do have a lot of bugs in their software packages.[22]

In my thesis work I have focused on the code refactoring level. On the basis of my research I have found that till now most of the refactoring work has been done on languages like Java, C++, HTML, PHP and Ruby. Seeing the growing popularity of Javascript[5] I thought about ways to apply these refactoring techniques to the Javascript code. I have also tried to figure out why some of the techniques applicable to Java/C++ can't be applied to Javascript.

This thesis describes the way in which refactoring techniques can be applied to Javascript code in a way that preserves the external behavior of a software. These techniques can prove to be beneficial in the design, reuse and evolution of object-oriented Javascript code.

The focus of this thesis is on the ways in which the refactoring techniques applicable to Javascript code that can be automated completely or up to a certain level. There are certain language specific syntactic constraints that need to be satisfied to automate some refactorings. I have discussed them in my thesis.

This thesis also discusses limitations of *class based object-oriented languages* like Java and C++ over *prototype based object-oriented languages* like Javascript that led me to choose Javascript as the preferred language. The need for refactoring object-oriented code has also been discussed till a certain depth. Several algorithms have been developed by me during this thesis work which I have tried to implement practically in my tool AKAAR[60] for refactoring Javascript.

Some of refactorings discussed in this thesis might seem practically not too difficult to many people but these refactorings require details regarding interrelationships between various parts of a program.

List of Figures

<u>List of Tables</u>

Table Of Contents

Chapter 1

Introduction

Web development teams have been operating in the dark for far too long. The lack of proven development methodologies for the Web environment has resulted in a constant struggle for developers to produce quality Web-based projects on time and within budget. The field is multidisciplinary in character, involving both technology and graphic design: Web-based project development must address the issue of company image, must function on multiple platforms, and must incorporate multiple media into one complete package. It has been found that agile approaches for software development such as Extreme Programming (XP)[52][39] can be adapted and applied to the Web-based project development process. Many books[36] demonstrates how the hallmarks of XP[9]--continuous integration, short iterations, paired programming, automated testing, refactoring and extensive client involvement--are particularly well suited to the unique demands of Web-based development.

The impact of web applications on our lives cannot be neglected these days. Applications like Gmail[43], Facebook[44], YouTube[45], Picnik[46], Google Analytics[47], Blogger[48], Google Docs[49], Google Books[50] etc. have changed the way we used to process, manage and share our data. 90s was the age of desktop applications written in high level languages like C, Java etc. but, in the 21st century things have changed a lot with a boom in web technologies. With the introduction of HTML5 standard[51] for web applications things are going to get better in the coming years. Now we will be able to develop applications using Javascript that wasn't possible till now. Some of the popular applications developed by using object-oriented features of Javascript are popular code quality tools for Javascript such as JsLint[26], JsHint[29] etc. Moreover add-ons on Firefox[56] are also developed using Javascript.

Javascript as we all know is a client side object-oriented scripting language supported by almost all major web browsers like Firefox[56], IE[57], Opera[58], Chromium[59] etc. Though lot of work has been done on refactoring PHP, HTML etc. but still I wasn't able to find scholarly articles regarding refactoring techniques for Javascript.

Fabrice Bellard has written a 32 bit x86 emulator in Javascript that is capable of running Linux in our modern day web browsers in May 2011[61]. Hence we can see that now Javascript has matured almost to the level of old players in object-oriented domain like C++ and Java. We can't neglect Javascript just by considering it as a mere client side scripting language because now it is capable of doing many things that were beyond our imagination few years back. Thus refactoring of Javascript will play an important role.

1.1 Motivation

The concept of refactoring originated in the Smalltalk circles in early 1990s. It was introduced by a software developer Kent Beck then. Few years later another renowned member of object-oriented group Martin Fowler wrote a classical book[1] on this interesting topic. He has also maintained an online catalog for refactorings applicable to Java since then.[2] Since then the refactoring techniques mentioned in the book has been Integrated Development Environments like Eclipse, Netbeans etc. and frameworks including Visual Studio supports refactorings applicable to class-based object-oriented languages.

The research paper of year 2009 by M. Schäfer and O. Moor of Programming Tools Group, Univ. of Oxford, UK titled as *Of Gnats and Dragons: Sources of complexity in implementing refactorings*[35] mentions that very less work has been done by the research community in the domain of refactoring and still the possibilities of research in this unexplored topic remains almost infinite. Moreover it also mentions that still a lot needs to be done so that refactoring can be automated by tools in a correct fashion. IDEs and frameworks have several bugs in implementing these refactorings too.[22]

The research paper also discusses how difficult it is to implement a simple refactoring named as *Move Method* in case of Java. The writers of the paper have presented the complexity associated with automating refactorings in a in a perfect fashion.

Though a lot of work has been done on refactoring techniques in case of class-based object-oriented languages but I was not able to find adequate literature on refactoring techniques for prototype based languages like Javascript. These were the factors that motivated me to write this thesis work in case of Javascript as this language is gaining popularity these days in the web development domain due to introduction of HTML5 standard by W3C. Moreover some of its peculiar characteristics like:

- Being a prototype based rather than class based object-oriented language.
- Treating functions as objects.
- Keeping arguments in an array when passed during a function call. etc.

has motivated me to work on this interesting topic as these syntactic characteristics can help in developing new refactorings for Javascript like *Mention Parameters* and *Removing with() statement* in this thesis.

The complexity in automating refactoring techniques in refactoring tools and lack of refactoring tools for Javascript has also motivated me in developing an online refactoring tool AKAAR[60] for Javascript.

1.2 Related Work

Refactoring is a very complex topic in itself. Not many books are available on this complex subject till today. A classical book[1] has been written by Martin Fowler in 1990s and that has acted like a bible for all the development work that has been done in this field since then. The writer of this book has also maintained an online catalog for refactorings applicable to Java. There is a list of around 93 refactorings in that catalog maintained by Martin Fowler[2].

The refactorings mentioned in the books and catalog have been implemented in IDEs like Eclipse, Netbeans etc. and frameworks like Microsoft's Visual Studio etc. But a major problem with these IDEs and frameworks is that they have focused mainly on moving and renaming methods as far as refactoring is concerned. Though it is true that 90% of the refactoring that is done on a given code comprises of renaming and moving methods but we can't neglect the other refactorings like Consolidate Duplicate

Conditional Fragments, Split Loop etc. Moreover these IDEs and frameworks have several bugs in their implementations of refactorings.[22]

A research paper of year 2009 by M. Schäfer and O. Moor of Programming Tools Group, Univ. of Oxford, UK titled as *Of Gnats and Dragons: Sources of complexity in implementing refactorings*[35] mentions the fact that the refactoring techniques have been implemented in an ad-hoc fashion in these IDEs due to which a lot of people don't know how to automate these refactorings in a refactoring tool. Such ad-hoc approaches have contributed in increasing complexity of this interesting topic. This paper also discusses that syntactic constraints of a language i.e. Gnats can pose serious problems in automating refactorings via a refactoring tool for the language. The authors of this research paper have also discussed this factor by means of automating the *Move Method* refactoring in case of Java in their research paper.

Several surveys have been done in case of refactoring tools. One of them was published by *Tom Mens* and *Tom Tourwe* in IEEE transactions of software engg, 2004. The title of the paper was *A survey of software refactoring.*[69] This paper provides an extensive overview of existing research in the field of software refactoring at that point of time. The research in the paper has been compared and discussed based on a several criterias: the refactoring activities that are supported, the specific techniques and formalisms that are used for supporting these activities, the types of software artifacts that are being refactored, the important issues that need to be taken into account when building refactoring tool support, and the effect of refactoring on the software process.

1.3 Problem Statement

Many of the refactorings applicable in class-based object-oriented languages cannot be directly applied to prototype-based dynamic object-oriented languages like Javascript. In fact many of them like *Extract Class*, *Move method* and *Rename Method* need to be modified in case of Javascript due to its syntactic constraints.

A blog article by Douglas Crockford[23] who is a renowned programmer in the field of Javascript discusses that the with() statement in Javascript code can pose several problems in the understandability of code by a third party programmer during debugging or code analysis. A third-party developer reusing our code having with() statement may get confused in the fact that an object used inside the with() statement is a global object or the one belonging to the object hierarchy. This helps in developing a new refactoring technique.

Similarly Javascript is different from Java because Java can be used on client as well as server side but Javascript is always used on the client side. Hence many refactorings like *Remove Middle Man, Hide delegate* etc. can't be applied in case of Javascript.

Javascript also treats arguments passed to a function during a function call in a very unique fashion. It stores those arguments in an array called *arguments[].* Hence there is no need to mention parameters during a function definition. This leads to the need for mentioning parameters in case of Javascript.

Hence an appropriate problem statement for this thesis can be mentioned as following:

"Developing refactoring techniques for Javascript code and develop a tool that can automate such refactoring techniques."

During the thesis work I have tried to develop a refactoring tool for Javascript called AKAAR too.

1.4 Scope of Work

The term refactoring can be applied to several domains like databases, object-oriented designing languages like UML or class based OO languages like Java, C++ etc. But we are going to focus specifically on the code refactoring techniques that seem applicable for code written in Javascript.

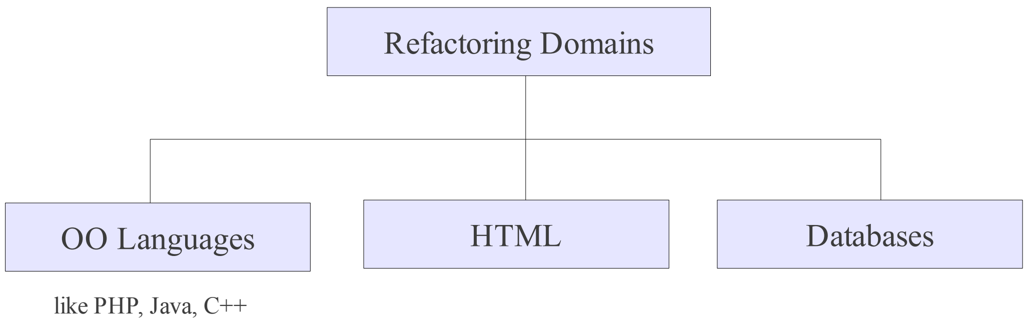


Fig. 1.1 Domains in which concept of refactoring has been applied

The scope of this thesis revolves around Javascript and its characteristics that can be utilized in discovering new refactoring techniques that can be applied to Javascript code. Moreover it also focuses on how existing refactoring techniques[1] can be applied to Javascript. As has been discussed by *Max Schafer and Oege de Moor* in their research paper[35] refactoring is a complex process that requires knowledge about the syntactic constraints of the language (called as *Gnats* in the paper). In this thesis we focus on some of the existing refactorings like *Rename Method, Consolidate Duplicate Conditional Fragments* etc. that can be applied to Javascript. Moreover this thesis also focuses on how syntactic characteristics of Javascript like passing of arguments in an array while calling a function can lead to proposal of new refactoring techniques like *Mention Parameters*. It also focuses on the some of the statements like *with()* in Javascript that may reduce the understandability of code over a long period. Focusing on such characteristics have lead to proposal of a new refactoring technique called *Remove with() statement* during this thesis work.

It also focuses on the specific characteristics of Javascript being a prototype based dynamic language in comparison to other counterparts like Java/C++ that are class based object-oriented languages.[10] This can lead to several modifications in the way existing refactoring techniques like *Rename Method*, *Extract Class* can be applied to Javascript.

This thesis also focuses on the need to embed a parser while developing a refactoring tool like AKAAR[60]. It also discusses the important role a parser plays in refactoring process. Though the book by Martin Fowler[1] merely discusses a few words about the parser but this thesis discusses and explains the role played by a language parser in a refactoring tool. In developing the tool AKAAR a parser called *jParser*[40] implemented by Tim Whitlock has been used.

1.5 Organization of Thesis

Chapter 1 of this thesis provides a brief introduction to the problem that I have tried to solve in this thesis. In order to work on a thesis topic there must be some kind of motivation behind it. It also discusses the motivation behind choosing such a complex topic and a specific language. Its too hard for a research fellow to implement and discuss each and every aspect of a given domain on which he/she decides to work. So there has to be a limited scope in which a researcher must do his/her work. The scope of this thesis work has also been discussed in section 1.4 of this chapter.

Chapter 2 of this thesis discusses refactoring and its applicability to web applications. Section 2.1 provides a brief introduction to the world of refactoring. Section 2.1.1 discusses types of refactorings. Section 2.1.2 lists few refactorings. Section 2.1.3 discusses how refactoring fits in software reusability. Section 2.1.4 discusses how we can make our code agile by using refactoring. Section 2.2 discusses pros and cons of refactoring or rewriting the code. Section 2.2.1 discusses why rewriting code is a chaos. Section 2.2.2 discusses why we need to prefer refactoring over rewriting of code. Section 2.3 discusses the need to refactor web applications. Section 2.4 discusses the advantages in refactoring the code.

Chapter 3 of this thesis discusses Javascript characteristics as well as refactoring techniques proposed by me for the language. Section 3.1 provides a brief overview about Javascript characteristics and browser support. Section 3.2 discusses a very important characteristic of Javascript i.e prototypes. It compares the approach for object orientation between classes and prototypes. Section 3.3 discusses some refactorings given in the refactoring catalog that are not applicable to Javascript. Section 3.4 discusses the *"Move Method"* tweak in case of Javascript. Section 3.5 discusses "Mention Parameters" refactoring that has been proposed by me. Section 3.6 discusses the *"Remove with() statement"* refactoring proposed by me.

Chapter 4 of this thesis deals with implementation issues that need to be considered while understanding how I have tried to implement the refactoring techniques in the refactoring tool AKAAR. Section 4.1 deals with technical specifications for the tool AKAAR. Section 4.2 explains the overall directory structure of tool. Section 4.3 deals with the process behind parsing of Javascript and a small demonstration of output of jParser. Section 4.4 deals with how I have embedded and reused jParser in AKAAR. Sections 4.5 – 4.8 explains how I have implemented these refactorings in my tool AKAAR.

Chapter 5 discusses the results that I have achieved in proposed and old refactorings from my tool AKAAR.

Chapter 6 deals with conclusion of this thesis. It summarizes its contributions and discusses the limitation in my approach while working on "Refactoring Techniques for Javascript".

At the end of this thesis we have a section for references and an appendix for regular expressions terminology.

Chapter 2

Refactoring and its significance in web apps

A software project is capable of becoming a monster of missed schedules, blown budgets and flawed products. One approach to achieve meaningful reductions in software costs is to acquire an existing software system rather than developing a new one from scratch. Often, however, the available software systems do not provide an exact solution for the problem at hand. Software that solves a similar problem might be available, but such software may need to be modified in some way before it can be reused. These changes may involve restructuring (in our case refactoring) the software.

2.1 Introduction to Refactoring

In most cases it is not possible for a software developer to have a perfect design prior to coding phase. In a software engineering[8] course a student is always taught that a good design comes first and coding comes second. But in reality what happens is that over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The coding activity gradually sinks from engineering to hacking.

Refactoring is the opposite of this practice. With refactoring we can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic.

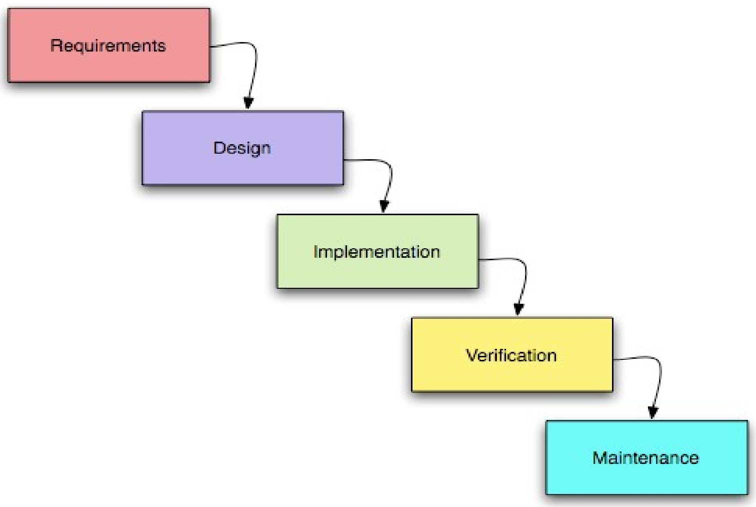


Fig. 2.1 Steps in software development life cycle

In many of the cases it will happen that the quick and dirty solution will work well for a particular problem. The compiler will never care whether the code is ugly or clean. But when we change the system, there is a human involved, and humans do care. A poorly designed system is hard to change. Hard because it is hard to figure out where the changes are needed. If it is hard to figure out what to change, there is a strong chance that the programmer will make a mistake and introduce bugs.

2.1.1 Types of Refactorings

Broadly speaking refactoring can be categorized into two categories.[1]

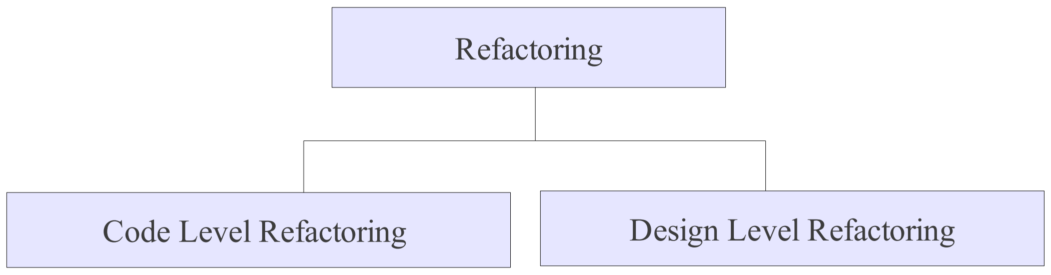


Fig. 2.2 Classification of refactoring on the basis of levels

NOTE: Extreme Programming guidelines asks a team to refactor mercilessly at design level.

But there is also another categorization because of recent advanced approaches that have resulted in runtime refactoring of a software. One of the approach for achieving this is the one by using a middleware.[13] The classification of refactoring has thus been classified on these parameters in the next diagram.

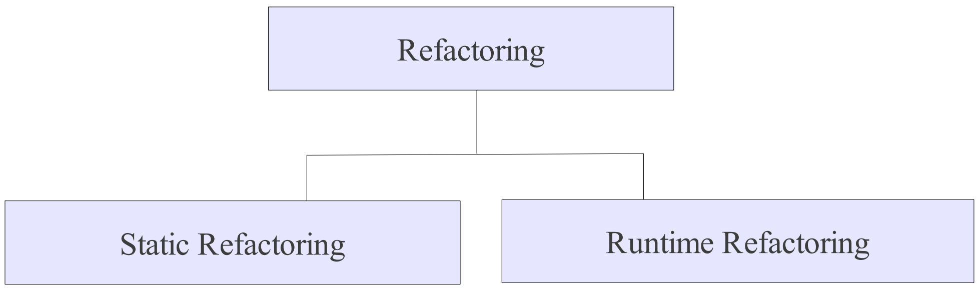


Fig. 2.3 Classification of refactoring on the basis of phases

2.1.2 Foundations of Refactorings

Refactoring is all about cleaning up code in an efficient and controlled manner. According to the book published by Martin Fowler et al.[1] and research paper published by Martin Fowler[3] in proceedings of the 24th International Conference on Software Engineering in the year 2002 the word *Refactoring* has been given two definitions depending on context.

Refactoring (noun): *A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

Refactor (verb): *To restructure software by applying a series of refactorings without changing its observable behavior.*

There is a catalog maintained by Martin Fowler[2] for refactoring techniques applicable to Java. Following is the list of refactorings applicable to class-based object-oriented languages like Java[2]:

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional

- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static Construction
- Convert Static to Dynamic Construction
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method

  ……....................

  ……....................
- Split Temporary Variable
- Substitute Algorithm
- Use a Connection Pool
- Wrap entities with session

Most of the work that has been done in the field of refactoring has been based on these techniques. IDEs and Frameworks have embedded refactoring capabilities in their products in an ad-hoc fashion. But Martin Fowler has clearly mentioned in his paper[1] that still many new refactorings can be proposed for other languages due to syntactic constraints associated with refactoring techniques.

2.1.3 Software Reuse and Refactoring

The high costs of developing software motivate the reuse and evolution of existing software. Software reuse in its broadest sense involves reapplying knowledge about one software system to reduce the efforts of developing and maintaining another system. Closely related to software reuse is software maintenance, where knowledge about a software system is used to develop a version that refines or extends it.

Approaches that support reuse address one or more of the following four important aspects :[11]

1. *finding* a reusable component. This is usually is not as simple as finding an exact match, but rather involves finding the most similar component.
2. *understanding* the component. Understanding what a component does is important in order to use it, but developing that mental model is difficult.
3. *modifying* a component or a set of components. Understanding what changes are needed has proven to be human-intensive and few tools have proven very helpful. These modifications often involve restructuring (or refactoring).
4. *composing* the components together. The composition process can be difficult, especially when a component has the dual purposes of being a useful independent entity and being used to create other composite structures.

2.1.4 Make Code Agile by Refactoring

Definition of refactoring[1] states that:

“Refactoring is improving code without changing the features it implements.”

While we are refactoring, we do not fix bugs, we do not improve performance of the software and we do not increasing robustness of code. Refactoring simply improves the design of the code, while ensuring that it still works the same way.

IT project managers throughout the world do not like hearing such things because according to them this leads to wastage of time during tight schedules for software development. They think that the business value of the software stays the same but the cost to the business goes up. But the truth is that it's not like that.[16]

If we measure business value only in terms of features that we have today in our software, then we can end up deep in technical debt soon. We can add features today in such a way that features tomorrow cost more and more. The value of refactoring is that it provides the future ability to programmers to comprehend and modify the code easily. This is called maintainability, but we can call it *Agility*. Code that has been properly refactored is agile code because it can be modified in future quite easily.

If we look in terms of economics then refactoring is like an investment or like repayment of a debt.

The term refactoring actually comes from mathematics. Following is a simple algebraic expression from high school algebra:

$$5y^3 + 10y^2 + 5y$$

If we refactor the expression by extracting the common factor, 5y, we get the following expression:

$$5y.(y^2+2y+1)$$

Sometimes it may become hard to spot the common factors both in the case of mathematics and programming. But sometimes it may also be done poorly.

The primary purpose of refactoring is to control complexity. Complexity and scale in codebases is a major contributor that leads to schedule blowouts, poor velocity and excessive development cost.

We can categorize complexity into two parts as *Essential complexity* and *Accidental complexity*.

Essential complexity can be marked as the complexity in the domain for which we are developing a software. We can divide and isolate essential complexity but we can never remove it. Essential complexity belongs to the problem in hand. By contrast, accidental complexity is associated with languages, frameworks and systems we use for development. Accidental complexity can be reduced by changing the system. Accidental complexity belongs to the solution. Refactoring helps in reducing accidental complexity.

One of the most basic refactoring techniques is *Extract Method* which all decent IDEs like Eclipse, Netbeans etc. can easily automate. We know we need Extract Method if we have a multi-page method with a sequence of of comment blocks. If Doing it manually means snipping out a logical sequence of code and pasting it into a new, small, well-named method, stitching the local variables used from the originating context into parameters to the method. If this is hard due to sloppy scoping or too many variables, we may consider Introducing a Field from a local variable.

The inner loop of agile development should go like this: Red, Green, Refactor. Red means you have a test which is not passing. Getting the test to pass is the next step. Green means you are passing all tests. Refactor means... refactor.

Even if we're a good agile developer, doing things as simply as possible, complexity and duplication of common factors creeps in while you're trying to pass tests. Everybody hacks. Everyone copies and pastes. This is fine as long as we go back and refactor when we've got the green bar.

Unit tests are really important for refactoring. If we're not doing unit testing we've got a long way to go. A good unit test suite is a necessary precondition for confident, aggressive refactoring. And IMHO a good type system is a necessary precondition for confident, aggressive, automated refactoring. These preconditions can present a quandary for some developers. Legacy systems often have no effective automated tests. And since they're often composed entirely of spaghetti, they need to be refactored. It's a chicken and egg situation, where do you start? All I can say here is we must start small.

A common question being asked quite frequently is that "Can you have too much refactoring?" Absolutely. If we're somewhere around middle-stage zealotry for this refactoring stuff, we may not be in danger of copy+pasting our way to a big ball of mud, but we may fall prone to exceed the safe working abstraction load of our language or go too far beyond the idioms of our team's codebase or comfort.

Every language has limits imposed by its design and implementation.[35] In Java and C#, for example, the limits are seen by many in the dynamic languages camps to be too much to bear. For example, say we're refactoring some Java or C# code. We might create a new interface with a few alternate concrete implementations and, whereas before we had two methods on a concrete class and a few big if-else blocks, after refactoring we might have three files and more actual lines of source code. It can be somewhat subjective but sometimes we may have more complexity even though we've removed duplication!

If this happens we have fallen asleep on the refactoring train and missed our station. Often we should just roll back the code and go write a feature. Some duplication is easy to see and cope with, especially if it can fit on one screen and any reader can see the pattern. In other languages, Lisp comes to mind, there are constructs (like macros) which allow us to encapsulate expressions that cannot be elegantly factored in, say, Java.

So the expressiveness of the language can constrain refactorability. Another way of saying this is that the language contains accidental complexity and only factoring out the language can remove that complexity.

As a more concrete example, lexical closures are a great way of implementing things like the new for loop (for each) introduced in Java 1.5 and functor frameworks that employ anonymous inner classes in Java for similar purposes (e.g. composable transformers, ad hoc iterator delegators instead of explicit looping) often feel too cumbersome compared to most closure implementations. So we just have to suffer the duplication and code bulk.

So in summary, Red, Green, Refactor, don't go overboard and be aware when our language makes capturing factors we see in your system worse. So the aim is simply to kill the complexity before it kills us.

2.2 Refactoring and Rewriting

Code base of a large project gets worse over time. I hope there are lucky exceptions, but in general it is true for most projects. The reasons are quite obvious:

- **More and more features**. It leads to increased complexity.
- **Shortcuts and hacks** to support “We need this fancy search till August. Period!” features

- **Developers rotation**. New developers don't know all the fundamental decisions and ideas behind the architecture. Knowledge gets lost with transition inevitably.

- **Development team growth**. More people – less communication. Less communication – bad decisions. It leads to code duplication, hacks to make something work without deep understanding of the underlying conditions etc.

Suddenly we can't add new features easily, we can't make significant changes easily, we have tons of technical debts and development team is close to bankruptcy. We want to change that and have just two options: refactoring or rewriting everything from scratch.

2.2.1 Rewrite and Chaos

When we rewrite from scratch, we add such a large portion of chaos that it is hard to predict the final result. We have a new singularity that will explode to the new product universe. But are we certain that it will be better than the previous universe? How many the 'same bugs' will we fix in the new product version?

However, rewrite may look faster. I mean we may release a new version faster with rewrite, but most likely with more bugs and less stable.

I think we may expect the results to be something like that:

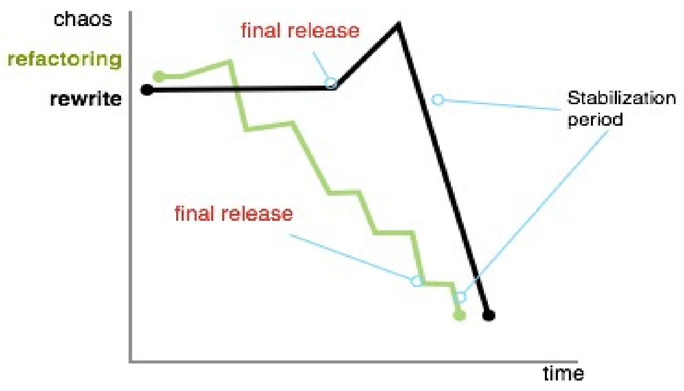


Fig. 2.4 Rewrite vs Refactor

The green line shows how chaos changes with refactoring. After each refactoring there is a small increase of chaos, but then the system becomes stable and chaos decreases. We can see that the final release is quite late, but we must keep in mind that there have been many releases before, so customers benefit earlier.

Black line is how chaos changes with a full rewrite. We have the old system during rewrite, so chaos is constant. After the public release chaos increases significantly. Quite many new (and old) bugs and quirks are expected, so stabilization period is longer. But the release itself is faster. The reason is that there is no burden behind. For example, there is no need to support all the places when we do a change, while in refactoring it is required to keep system working and stable all the time, and it demands additional effort.

"Refactoring is adaptation, while full rewrite is revolution. Again, revolution is a chaotic beast. We may slowly adapt the product for new external conditions or make one revolutionary rewrite."

2.2.2 What to choose Refactoring or Rewriting?

I do think there is a unique answer to this question. I think it will depend on the situation a development team is. If time to market the software is very important, if there is a chance that we can loose business if a new version will not be published in 3 months, then we shall try a full rewrite of the software to be developed. But one should be aware of its side effects too! It may lead to significant drop in quality and long stabilization period may hurt existing customers of our organization.

In most cases refactoring is the preferred choice. Slow pace, happy customers, constant improvements, high quality.

2.3 Refactoring and Web Applications

As I have already mentioned that during 90s desktop applications developed in Java and C++ were popular among users but with a boom in web technologies in the last decade the scenario has changed abruptly. Now developers are developing web applications that are substitutes of desktop applications that were developed few years back. One of the best example of such an application is *Google Docs[49]*. It has been developed by Google as a substitute for desktop office suites like *Microsoft's Office 2010[62]* and *Oracle's OpenOffice[63]*.

A web application is not developed in a single language. It is actually composed of several components written in different languages like HTML[64], CSS[65], PHP[66], Javascript[5], JScript etc. Each of these languages are meant for specific purpose. For example, Javascript is meant to be used as a client-side scripting language while PHP is meant to be used as a server side scripting language. Hence we can't neglect the impact these languages have in a web application. Moreover the concept of reusability is prevalent among web developers. Scripts written in these scripting languages are reused in other projects too. Hence I believe that refactoring of Javascript will play a major role in developing high quality web applications.

Refactoring can also play a major role in improving usability of web applications. This topic has been discussed quite well by Gustavo Rossi et al. in their research paper.[17]

2.4 Why we need to Refactor?

I don't want to proclaim refactoring as the cure for all software ills. We can't consider it as some sort of "silver bullet." We can't say that refactoring our code will improve its performance in terms of complexity metrics.[18] Yet it is a valuable tool, a pair of silver pliers that helps us keep a good grip on our code. Yi Wang in his research paper[19] has clearly mentioned the factors that motivates a developer to refactor. Refactoring of Javascript is necessary for the following reasons:[1]

- *Refactoring improves the design of software.*

  Without refactoring, the design of the program will decay with time. When people change code to realize short-term goals, the code loses its structure because generally changes are made without a full comprehension of the design of the code. It becomes harder to see the design by reading the code.

Refactoring is rather like tidying up the code. Work is done to remove bits that aren't really in the right place. Loss of the structure of code has a cumulative effect. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring helps code retain its shape.

One of the refactoring technique implemented in AKAAR[60] for this purpose is *"Consolidate Duplicate Condtional Fragments"*.

- *Refactoring makes software easier to understand.*

  The trouble is that when we are try to get the program to work, we do not think about that future developer. It takes a change of rhythm to make changes that make the code easier to understand. Refactoring helps us to make our code more readable. When refactoring we must have code that works but the code may not be ideally structured. A little time spent refactoring can make the code better communicate its purpose. Programming in this mode is all about saying exactly what we mean.

  One of the refactoring that has been implemented in AKAAR[60] for this purpose is *"Rename Method"*.

- *Refactoring helps in finding bugs.*

  Help in understanding the code also helps spot bugs. Generally people are not terribly good at finding bugs. Some people can read a lump of code and see bugs but many can't. However, I find that if we refactor code, we work deeply on understanding what the code does, and we put that new understanding right back into the code. By clarifying the structure of the program, we clarify certain assumptions we've made, to the point at which even we can't avoid spotting the bugs.

- *Refactoring helps us program faster.*

  In the end, all the earlier points come down to this: Refactoring helps us develop code more quickly.

A good design is essential to maintaining speed in software development. Refactoring helps us develop software more rapidly, because it stops the design of the system from decaying. It can even improve a design.

Though there are several other applications of refactoring but I have focused on the above mentioned applications of refactoring during the thesis work. But still I will like to mention one of the important application of refactoring that has been exploited for parallelization of single tier applications:

- *Refactoring helps in parallelization of software*[14][15][25]

  Old programming languages and tool suites are designed for quick and easy construction of sequential, non-distributed applications. But we are now developing applications over distributed systems like World Wide Web.

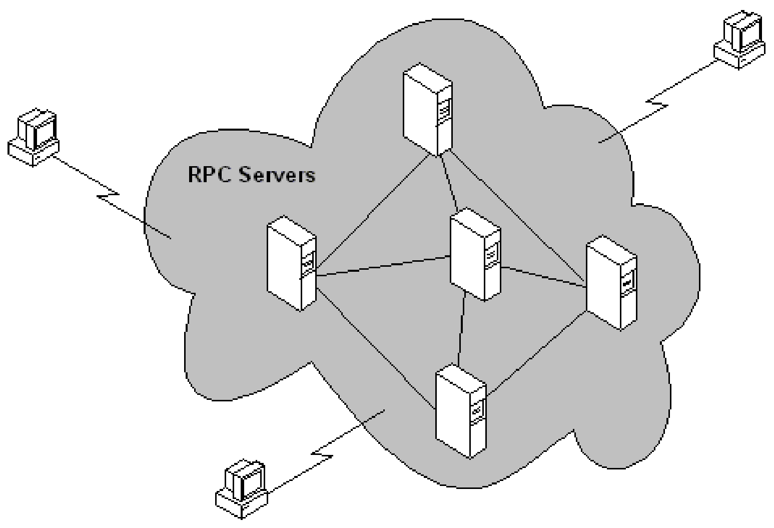


Fig. 2.5 Distributed System

In order to write distributed applications, programmers must learn and use a large variety of lower-level libraries for cross-tier communication, data marshaling, synchronization, and security. The libraries' sole purpose is to support distributed execution of application logic that could just as well be executed sequentially. In fact, programmers often create prototype applications that run in simplified, streamlined, sequential environments so that they can test and debug their code. Then they manually break up the prototype, inserting communication and synchronization code and distributing the pieces among multiple execution tiers.

Refactoring is about breaking the code into small pieces so that they can be easily handled. These chunks of code can easily be assigned remotely available resources over a distributed system. Hence refactoring can be deployed for parallelization of single tier applications.

NOTE: The last point has nothing to do with the refactoring techniques for Javascript.

Chapter 3

Javascript, its characteristics and Refactoring

3.1 Javascript Overview

The following table demonstrates the key characteristics of Javascript:[38]

Paradigm	Multi-paradigm: scripting, prototype-based, imperative, functional
Appeared in	1995
Designed by	Brendan Eich
Developer	Netscape Communications Corporation, Mozilla Foundation
Stable release	1.8.2 (June 22, 2009)
Preview release	1.8.5 (July 27, 2010)
Typing discipline	Dynamic, weak
Major implementations	KJS, Rhino, SpiderMonkey, V8, WebKit
Influenced by	C, Java, Perl, Scheme, Python
Influenced	Jscript, Jscript.NET, Objective-J

Table 3.1 Javascript Characteristics

Following table[38] demonstrates the datesheet of Javascript language and its support in various available browsers:

Version	Release Date	Equivalent To	Netscape Navigator	Mozilla Firefox	Internet Explorer	Opera	Safari	Google Chromium
1.0	Mar 1996		2.0		3.0			
1.1	Aug 1996		3.0					
1.2	June 1997		4.0-4.05					
1.3	Oct 1998	ECMA-262 1st edition / ECMA-262 2nd edition	4.06-4.7x		4.0			
1.4			Netscape Server					
1.5	Nov 2000	ECMA-262 3rd edition	6.0	1.0	5.5 (JScript 5.5), 6 (JScript 5.6),	6.0-11.0	3.0-5	1.0-10.0.666

Version	Release Date	Equivalent To	Netscape Navigator	Mozilla Firefox	Internet Explorer	Opera	Safari	Google Chromium
					7 (JScript 5.7), 8 (JScript 5.8)			
1.6	Nov 2005	1.5 + Array extras + Array and String generics + E4X		1.5				
1.7	Oct 2006	1.6 + Pythonic generators + Iterators + let		2.0				
1.8	June 2008	1.7 + Generator expressions + Expression closures		3.0				
1.8.1		1.8 + Native JSON support + Minor Updates		3.5				
1.8.2	June 22, 2009	1.8.1 + Minor updates		3.6				
1.8.5	July 27, 2010	1.8.1 + ECMAScript 5 Compliance		4	9			

Table 3.2 Javascript evolution and browser support

The above table clearly demonstrates that Javascript is still evolving with subsequent ECMA Script specifications[4].

3.2 Classes vs Prototypes

Software developers that work with Object Oriented programming languages are familiar with the concept of classes. As it turns out, that's common but it's not the only way to accomplish object orientation and the JavaScript language designers chose not to use the most common one. A classic paper on the topic Classes vs Prototypes was written by Antero Taivalsaari of Nokia Research Center.[10]

In this part we will try to figure out how JavaScript implements inheritance and how we can use it to build rich object hierarchies.

3.2.1 Class Based Programming

If you have previous experience with Object Oriented languages such as Java, C++, C# or Visual Basic, chances are that you have employed class-based inheritance. You may have even concluded that was the only way to write object oriented software.

In class-based inheritance, there's a clear distinction between the classes (or class objects) and the instances (or instance objects.) The classes define the behavior and structure of instance objects, which in turn simply contain instance data.

The examples below illustrate a class being defined and then used in two different class-based programming languages.

```
//java or C#

class Door{
 public void open(){
  //...code omitted
 }
}
Door frontDoor = new Door();
frontDoor.open();

'Visual Basic
Class Door
 Public Sub Open()
  '...code omitted
 End Sub
End Class

Dim frontDoor as new Door()
frontDoor.Open()
```

Another important characteristic of class-based programs is how inheritance is implemented. Each class that you create has a base class (or super class) explicitly or implicitly defined. Members of the base class will become available to the new class (the derived or inherited class.)

Let's expand our examples a little bit to show class inheritance.

```
//java

class SafeDoor extends Door{
 public void unlock(string secretCombination){
  //...code omitted
 }
}
SafeDoor safe = new SafeDoor();
safe.unlock("4-8-15-16-23-42");
safe.open();

'Visual Basic
Class SafeDoor
 Inherits Door
 Public Sub UnLock(ByVal secretCombination As String)
  '...code omitted
 End Sub
End Class
```

```
Dim safe as new SafeDoor()
safe.UnLock("4-8-15-16-23-42")
safe.Open()
```

3.2.2 Prototype Based Programming

Some languages choose to offer object oriented programming through mechanisms other than classes. One such mechanism is prototyping. JavaScript, Self, Lua, ActionScript, Agora, Cecil and many other languages are prototype-based.

Prototyping is a way to create objects by replicating another object (the so called prototype.) The new object may or may not have a link to the original object, depending on the language implementation. JavaScript maintains a link between the two as we will see shortly.

In prototype-based languages there's usually an operator to effect the object creation by copying another object. Surprisingly JavaScript does not offer such operator, which is often consider a design flaw of the language.

What we are looking for in such operator is a way to write the following code.

```
//attention, this is invalid syntax for Javascript
var BRAND_NEW_OBJ = object( EXISTING_OBJ );
```

Unfortunately, the object function above does not come with JavaScript. On the other hand, nothing stops us from creating our own implementation of that operator.

```
function object(original) {

function F() {}
 F.prototype = original;
 return new F();
};
```

3.2.3 Prototypal Inheritance

Let's now take a look at prototypes in action and create an object hierarchy. Hopefully this will clarify how prototype objects relate to the new objects that derive from them. Consider the following simple implementation of an vehicle object.

```
var vehicle = {

 wheels: 0,
 color: 'white',
 make: 'ACME',
 model: 'Unknown',
 year: 1998
}
```

We can easily derive a more specialized car from the vehicle object with the help of the object function we mentioned above.

```
var car = object(vehicle);
car.doors = 2;
```

The above code first creates the *car* object by linking to the existing *vehicle* object that is given to the object function. This link is represented by the arrow in the diagram below. After creating the *car* object, we add a new property called *doors* with the value of *2*.

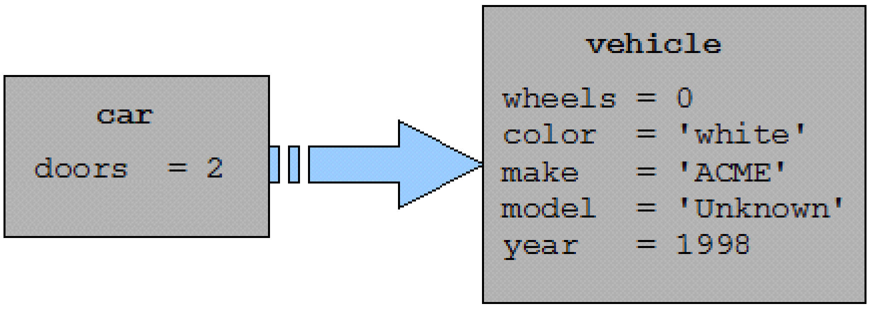


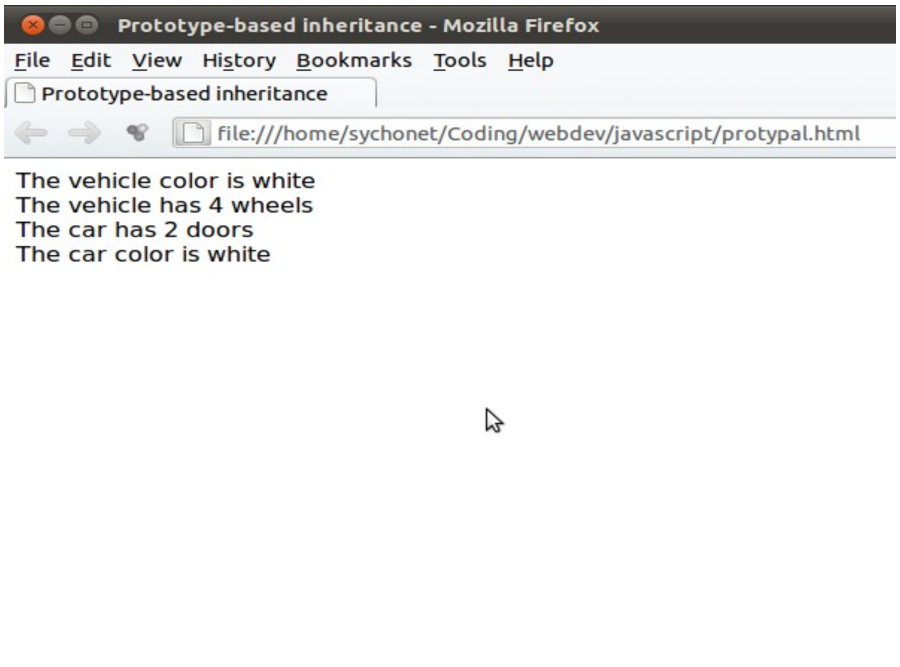
Fig. 3.1 Prototypal Inheritance

In the above diagram that car does not have a copy of all the properties from vehicle. Only the new *doors* property is stored in car.

Will this thing work? Let's try it on Firefox 4 web browser. Following is the code we are using for implementing prototypal inheritance.

```
<html>

<head>
  <title>Prototype-based inheritance</title>
 <script type="text/javascript">
 function object(original) {
  function F() {}
  F.prototype = original;
  return new F();
 };

 var vehicle = {
  wheels: 0,
  color: 'white',
  make: 'ACME',
  model: 'Unknown',
  year: 1998
 };

 var car = object(vehicle);
 car.doors = 2;
 </script>
</head>
<body>
 The vehicle color is
 <script
type="text/javascript">document.write(vehicle.color)</script>
 <br />
 The vehicle has
 <script
type="text/javascript">document.write(vehicle.wheels)</script> wheels
 <br />
 The car has
 <script type="text/javascript">document.write(car.doors)</script>
doors
 <br />
 The car color is
 <script type="text/javascript">document.write(car.color)</script>
 <br />
</body>
</html>
```

Following is the output


Prototype-based inheritance - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Prototype-based inheritance
file:///home/sychonet/Coding/webdev/javascript/protypal.html

```
The vehicle color is white
The vehicle has 4 wheels
The car has 2 doors
The car color is white
```

Fig. 3.2 Output of prototypal inheritance example

The first three properties that we print are not hard to understand. They are standard properties just like we have seen in previous lessons. The interesting line is the one that prints *car.color*. We did not explicitly add a color property to the car object, so when JavaScript interpreter executes and tries to find that property in car it won't. But the interpreter doesn't stop there. It will check if the object has a link to the object that was used during the creation process (the prototype object.) In our case car does have a prototype so the interpreter proceeds to that object,vehicle, and tries to find a member named color there. It will then find the property and print white.

The important thing to keep in mind here is that JavaScript knows about that arrow that we have in our diagram, and follows that arrow when something is not found in the object at hand. If the prototype object at the end of said arrow does not have the desired member, JavaScript will check if the prototype object has a prototype of its own and continue to do that recursively until the member is found or no more prototypes are available, in which case an error will be reported.

3.2.4 Dynamic nature of Javascript

A dynamic language is one where the type of the objects is only loosely bound to the way it was created. Objects may be created through a constructor method and inherit from a base class or a prototype but that doesn't mean that the object is locked for alterations.

Code can, at any time, add, remove or modify existing properties or methods of the objects. For that reason the base type of an object is less important in a dynamic language than it is in a statically typed language like Java or C#.

You may think that dynamic typing is a recipe for disaster but in practice that's not the case. There are two important factors that make dynamic languages very appealing.

Productivity

How many times when programming in statically typed languages we were in the situation where we had this class that was almost perfect but it lacked one important property or method? The usual route is to inherit a new class from that one and add the missing functionality.

That works well in the beginning but it quickly leads to class explosion in the application. Before you notice you'll have dozens of classes that are minor improvements over other existing classes.

Dynamic languages offer the capability of "fixing" the original classes on the spot and keep the code-base smaller and more manageable. That leads to greater programmer efficiency.

Unit Testing

Dynamic languages walk hand-in-hand with unit testing. We will take a closer look at unit testing in the next lesson but let's just say that automated unit testing will help detecting a bug at the moment it is introduced in the code.

I wouldn't try tell you that unit testing is very popular in JavaScript, but in other dynamic languages like Ruby and Python it's common practice in enterprise-quality software. JavaScript is in a way still discovering the importance of unit testing.

3.2.5 Declaring objects in Javascript

Objects are the fundamental unit of code encapsulation and reuse in any OO language. It is incredibly easy to create objects in JavaScript. There's even more than one way to do so.[5]

3.2.5.1 Simple method

The first approach is to create an empty object and progressively add its properties and methods.

```
var GUITAR = { };

   GUITAR.color = 'black';
   GUITAR.strings = ['E', 'A', 'D', 'G', 'B', 'e'];
   GUITAR.tune = function (newStrings) {
    this.strings = newStrings;
   };
   GUITAR.play = function (chord) {
    alert('Playing chord: ' + chord);
   };
   GUITAR.print = function (price, currency) {
    alert('This guitar is ' +
     this.color +
     ', it has ' + this.strings.length + ' strings' +
     ' and it costs ' + price + currency);
   };
   //using the object
   GUITAR.play('Dm7');
   GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
   debugWrite('this guitar is: ' + GUITAR.color);
   GUITAR.print(850, 'USD');
```

The above methodology isn't too hard to understand but it is certainly more work than we are used to in more popular programming languages. What we did here is quite simple. We just created the object and appended each property and method as desired.

3.2.5.2 Using Literal Notation

JavaScript also has a literal notation for objects. The previous example could have been rewritten in literal notation as follows.

```
var GUITAR = {

    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings) {
     this.strings = newStrings;
    },

    play: function (chord) {
     alert('Playing chord: ' + chord);
    },
    print: function (price, currency) {
     alert('This guitar is ' +
      this.color +
      ', it has ' + this.strings.length + ' strings' +
      ' and it costs ' + price + currency);
    }
   };

   //using the object
   GUITAR.play('Dm7');
   GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
   debugWrite('this guitar is: ' + GUITAR.color);
   GUITAR.print(850, 'USD');
```

The syntax is easy to understand. It is a comma-delimited list of *name:value* pairs. Note that the method declaration is easy to be confused with a regular function declaration. A function can be used as a value and that's what is happening here. We can also think of the methods as properties that contain a function as their values, if that helps us understand the notation.

JSON

JavaScript Object Notation, or JSON, is a subset of the literal notation that we just saw. JSON was first proposed by **Douglas Crockford** as a neutral way to represent and transport data, usually replacing XML.

JSON, just like the literal notation, is also a list of name/value pairs. The main difference is that the values can only be a string, an Array, a Number, true, false, null, or another JSON object. The field names are also enclosed in double-quotes.

Here's the GUITAR object represented in JSON. Note that we cannot represent the methods because JSON doesn't accept them. It makes sense because JSON is meant only for data interchange, where behaviors are irrelevant.

```
var GUITAR = {

 "color":"black",
 "strings":['E', 'A', 'D', 'G', 'B', 'e']
};
```

3.2.5.3 Using Factory Functions

One important thing to notice in the previous two approaches is that we did not need to create a formal class to serve as the template of the GUITAR object. If we needed a second guitar object we would need to create it the same way we did for the first one.

We could just encapsulate that logic in a function that can create a brand new guitar object on demand.

One important thing to notice in the previous two approaches is that we did not need to create a formal class to serve as the template of the GUITAR object. If we needed a second guitar object we would need to create it the same way we did for the first one.

We could just encapsulate that logic in a function that can create a brand new guitar object on demand.

```
    function createGuitar(color, strings) {
```

```
    var guitar = { };
    guitar.color = color;
    guitar.strings = strings;
    guitar.tune = function (newStrings) {
     this.strings = newStrings;
    };
    guitar.play = function (chord) {
     alert('Playing chord: ' + chord);
    };
    guitar.print = function (price, currency) {
     alert('This guitar is ' +
      this.color +
      ', it has ' + this.strings.length + ' strings' +
      ' and it costs ' + price + currency);
    };
    return guitar;
   }

   var GUITAR1 = createGuitar('black', ['E', 'A', 'D', 'G', 'B', 'e']
);
   var GUITAR2 = createGuitar('maple', ['F', 'Bb', 'D#', 'G#', 'C',
'f'] );
```

3.2.5.4 Using Constructors

There's a variation of the factory function methodology that may feel more natural. In JavaScript, when function is called preceded by the *new* operator, the function receives an implicit this argument that is a brand new object, ready to be assembled with properties and methods. Also, if we do not return anything explicitly, the *new* operator automatically returns *this*.

Let's rework our last example into a constructor. A good convention is to start constructor functions with a capital letter, to differentiate from a regular function, signaling to the programmer that it needs to be called with the *new* operator.

```
function Guitar(color, strings) {

    this.color = color;
    this.strings = strings;
    this.tune = function (newStrings) {
     this.strings = newStrings;
    };
    this.play = function (chord) {
     alert('Playing chord: ' + chord);
    };
    this.print = function (price, currency) {
     alert('This guitar is ' +
      this.color +
      ', it has ' + this.strings.length + ' strings' +
      ' and it costs ' + price + currency);
    };
}

var GUITAR =  new Guitar('black', ['E', 'A', 'D', 'G', 'B', 'e']);
debugWrite('this guitar is: ' + GUITAR.color);
GUITAR.play('Dm7');
GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
GUITAR.print(850, 'USD');
```

3.3 Some refactorings not applicable to Javascript

There are certain refactorings mentioned in Martin Fowler's catalog of refactorings[2] that can't be applied to Javascript. The reason behind this is that there are certain language specific features in case of a programming language that many refactoring techniques applicable in one programming language can't be applied to some other programming language. The following subsections of this section discusses such factors:

3.3.1 Extract Interface

The concept of interfaces in Java is an important one. This refactoring technique has been defined in book[1] as:

“If several clients use the same subset of a class's interface, or two classes have part of their interfaces in common then extract the subset into an interface.”

The basic concept can be grasped from the next figure.

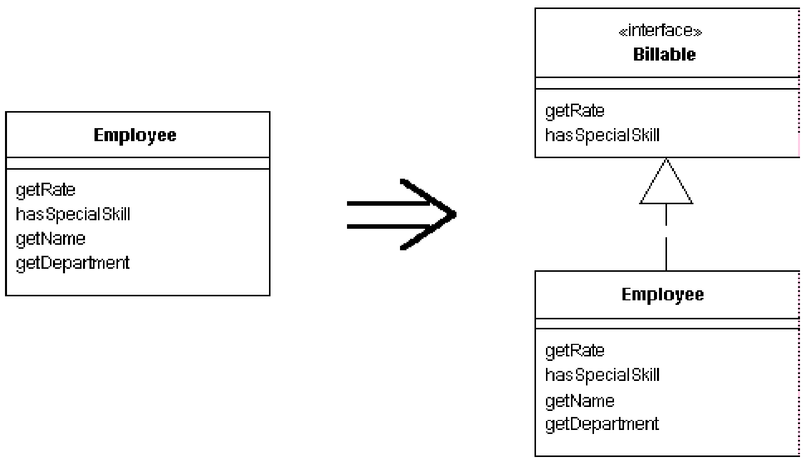


Fig. 3.3 Extract Interface

But in case of Javascript this refactoring technique can't be applied because Javascript doesn't support interfaces[67].

3.3.2 Hide Delegate

This refactoring has been defined in the book[1] as:

"A client is calling a delegate class of an object so create methods on the server to hide the delegate."

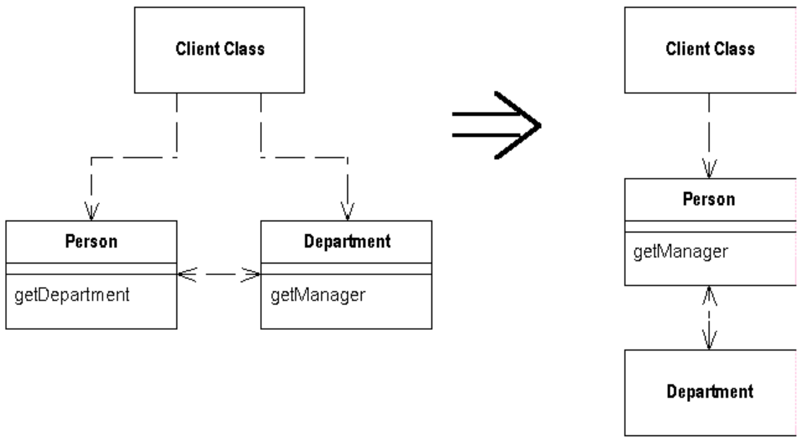


Fig. 3.4 Hide Delegate

Though Javascript can be applied at both server[68] and client ends but it is not a server side scripting language. As server side scripting is handled by scripting languages like PHP etc hence this refactoring technique doesn't make any sense in case of refactoring Javascript.

3.3.3 Remove Middle Man

This refactoring is exactly the opposite of the last refactoring. It states that:

"A class is doing too much simple delegation so get the client to call the delegate directly."

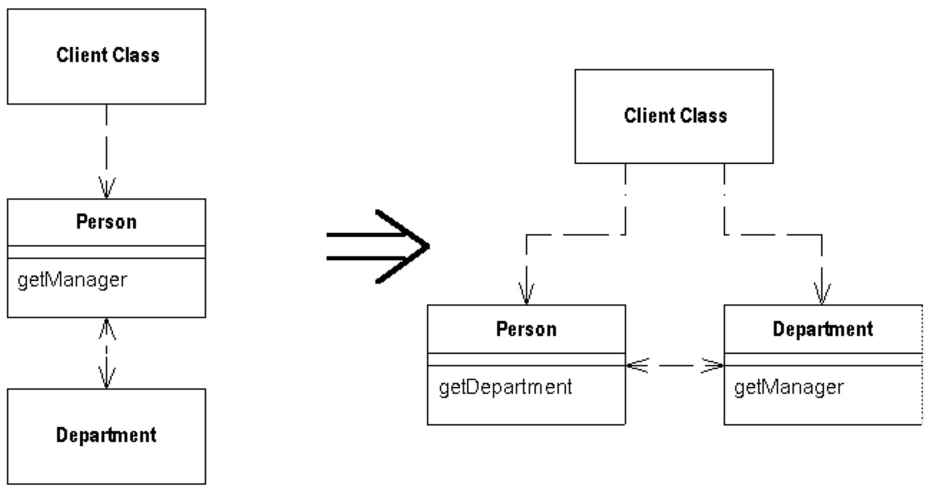


Fig. 3.5 Remove Middle Man

As the last refactoring is not applicable to Javascript so is the same reason why this refactoring is also not applicable to Javascript.

3.4 Move Method

In the book by Martin Fowler[1] this refactoring has been defined as:

"A method is, or will be, using or used by more features of another class than the class on which it is defined. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether. "

It has also been represented graphically as:

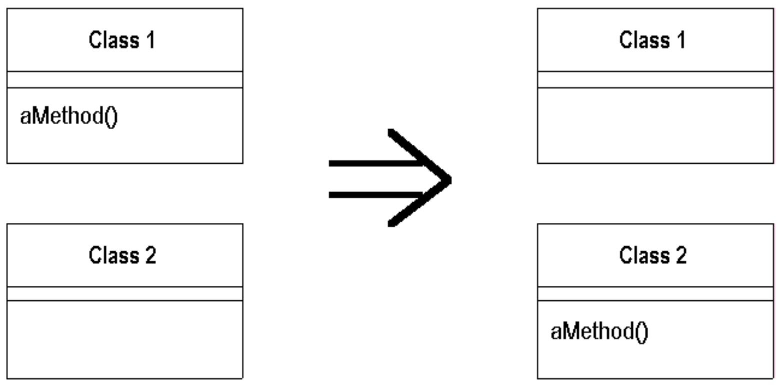


Fig. 3.6 Move Method

Following example will make the things more clearer.

```
class Project {
  Person[] participants;
}

class Person {
  int id;
  boolean participate(Project p) {
    for(int i=0; i<p.participants.length; i++) {
          if (p.participants[i].id == id) return(true);
    }
    return(false);
  }
}

... if (x.participate(p)) ...
```

After applying the move you end up with

```
class Project {
  Person[] participants;
  boolean participate(Person x) {
    for(int i=0; i<participants.length; i++) {
          if (participants[i].id == x.id) return(true);
    }
    return(false);
  }
}

class Person {
  int id;
}

... if (p.participate(x)) ...
```

As we can see that in case of Java and other class based Object Oriented languages like C++ etc in order to move a method from one class to another we have to move the entire definition/function body of the function from one class to another. But, in case of Javascript things change drastically. A key feature of Javascript is that every function is considered as an object in case of Javascript. This gives functions the capacity to hold executable code and be passed around like any other object.

```
var sum = function(){
     …....................
     …....................
     }

s.prototype.compute_sum = sum;
```

So in the above case we have declared a function object named as *sum* and we have associated this function with function object named *s*. Now suppose we want this function to be move to object *q* then all we have to do is to write one line of code.

```
var sum = function(){
     …....................
     …....................
     }

q.prototype.compute_sum = sum;
```

And the function sum will now be associated with function object *q* rather than function object *s*.

3.5 Mention Parameters

In the book by Martin Fowler[1] the refactoring *Add Parameter* has been defined as:

"Add a parameter for an object that can pass on this information. "

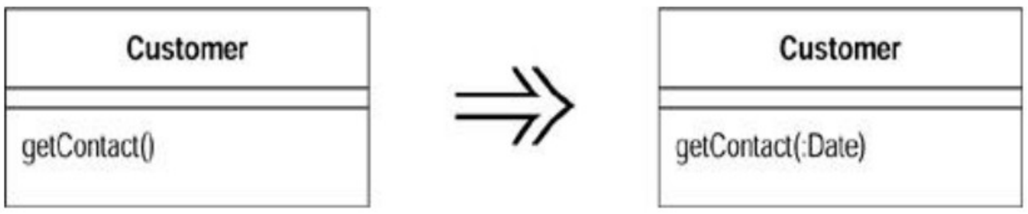


Fig. 3.7 Add Parameter

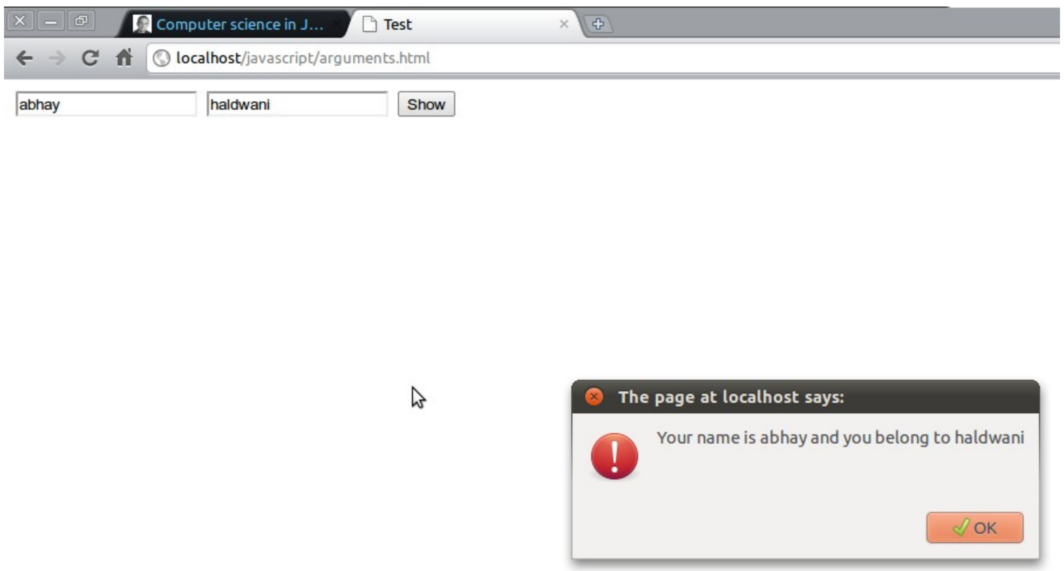
But a difference between Javascript and some other object-oriented languages like Java is that Javascript deals with arguments in a different manner. In case of Java the arguments being passed while the function is called must have corresponding counterparts in the function definition. For example,

```
class Customer
{
      private int cust_class;
      public void setClass(int class)
      {
            cust_class = class;
      }
      public static void main(String args[])
      {
            setClass(2);
      }
}
```

But in case of Javascript the situation is completely different. Here it is not necessary to mention the corresponding arguments being passes during the function call as parameters in function definition. For example, look at the following code

```
<html>

<head>
      <title>Test</title>
      <script type = "text/javascript">
            function show()
            {
                  alert("Your name is " + arguments[0] + " and you
                  belong to " + arguments[1]);
            }
      </script>
</head>

<body>
      <input type = "text" id = "Name" />
      <input type = "text" id = "Place" />
      <input type = "button" value = "Show" onClick =
"show(document.getElementById('Name').value,document.getElementById('
Place').value)"/>
</body>

</html>
```

The output for the above code will be as following



abhay

haldwani

The page at localhost says:

Your name is abhay and you belong to haldwani

Fig 3.8 Output while using arguments[]

So we can see that in case of Javascript there is no need to mention the corresponding parameter in function definition for arguments being passed when the function is called at a particular location. Actually arguments are treated like an array in case of a Javascript program.

```
argument[0], argument[1],. . . . . . . . . . . . ., argument[N]
```

Hence in order to enhance the understandability of such a kind of code I am proposing a refactoring technique for Javascript in this thesis.

```
function show()
{
      alert("Your name is " + arguments[0] + " and you belong to " +
      arguments[1]);
}
show(“Abhay”,”Haldwani”);
```

```
function show(name,place)
{
      alert("Your name is " + name + " and you belong to " + place);
}
show(“Abhay”,”Haldwani”);
```

Name: Mention Parameters

Summary: If arguments passed to a function during its call have not been mentioned in its definition as parameters then mention them explicitly.

Motivation: After a certain amount of time it becomes hard for even the programmer to understand a program written by him/her. A major role played in such a case is played by temporary variables and parameters. Arguments being passed to a function while it is being called at some other part of program must be clearly mentioned in the parameter list of a function in order to avoid understandability problems at a later stage while doing code analysis. As Javascript deals with arguments in a bit different manner when compared to other languages as shown above. Hence in order to preserve understandability of code its better to mention arguments passed during a function call as parameter in its parameter list definition.

Mechanism:

(Step1) Search for functions that when called in some other part whose definitions do not contain the corresponding parameters list for arguments passed.

(Step 2) Add corresponding parameter list in the function definition and replace each arguments[n] used in function definition with the nth parameter name in function definition.

3.6 Removing with() statement

I read about disadvantages of using *with()* statement in Javascript code. So I thought that this fact can be used to derive a language specific refactoring technique for Javascript. In this section of this thesis I am proposing a formal representation for the refactoring technique as have been described in the book by Martin Fowler[1].

Name : Removing with

Summary : If your Javascript code contain global variables then remove the with statement used in your program.

Motivation :

JavaScript's *with* statement was intended to provide a shorthand for writing recurring accesses to objects. So instead of writing

```
ooo.eee.oo.ah_ah.ting.tang.walla.walla.bing = true;
ooo.eee.oo.ah_ah.ting.tang.walla.walla.bang = true;
```

We can write

```
with (ooo.eee.oo.ah_ah.ting.tang.walla.walla) {
    bing = true;
    bang = true;
}
```

That looks a lot nicer. Except for one thing. There is no way that we can tell by looking at the code which *bing* and *bang* will get modified.

Will *ooo.eee.oo.ah_ah.ting.tang.walla.walla* be modified? Or will the global variables *bing* and *bang bang* get clobbered? It is impossible to know for sure.

The *with* statement adds the members of an object to the current scope. Only if there is a *bing* in

ooo.eee.oo.ah_ah.ting.tang.walla.walla

will

ooo.eee.oo.ah_ah.ting.tang.walla.walla.bing

be accessed.

If we can't read a program and be confident that we know what it is going to do, we can't have confidence that it is going to work correctly. For this reason, the *with* statement should be avoided.

Mechanism:

Define a *var*.

```
var o = ooo.eee.oo.ah_ah.ting.tang.walla.walla;
o.bing = true;
o.bang = true;
```

Now there is no ambiguity. We can have confidence that it is
ooo.eee.oo.ah_ah.ting.tang.walla.walla.bing
and
ooo.eee.oo.ah_ah.ting.tang.walla.walla.bang
that are being set, and not some hapless variables.

Chapter 4

Implementation Considerations for AKAAR

4.1 Technical Details

The tool AKAAR[60] has been developed using open standards and open sourced components. In other words it has got no license issues associated with it. The tool has been developed using *DHTML* (i.e. HTML 4.0, CSS 2 and Javascript) and *PHP 5.0*. The web server used in the background is *Apache 2.2.16*. The *PHP module* has been embedded with the web server so that the *.php* pages can be preprocessed at the web server's end. Some of the other tools used for debugging purpose are *Firebug* and *Firefox 4.0's Error Console*. The tool is currently available on the Internet at a server with our registered domain name of *"farzighumakkad.com"*. All the files have been uploaded in the document root of that server i.e. */var/www/*. At present the web application AKAAR is a single user application. I have put this constraint on it due to lack of resources but it can be easily transformed in a multi user web application by modifying few lines of its code.

4.2 Overall directory structure of the tool

The overall directory structure of the tool on the server on which the web application *AKAAR* has been hosted is as shown below. The same structure has been followed in the soft copy of the software that has been written by me in the CD attached at the back of the hard copy of this thesis.

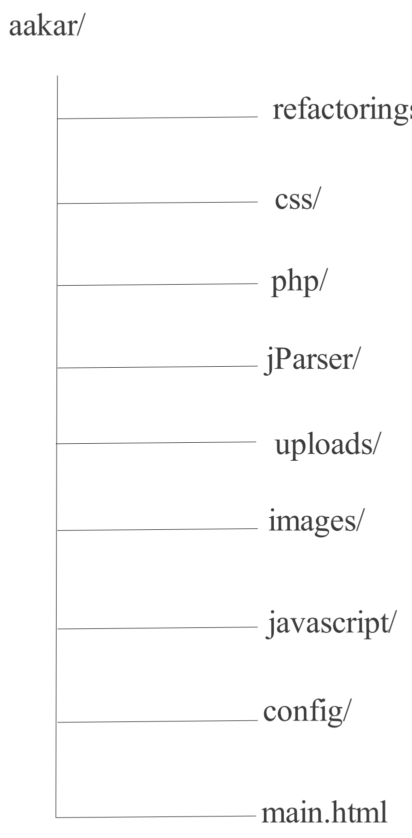


Fig. 4.1 Overall directory structure of AKAAR

The contents of various directories are as following:

- **refactorings/** : It contains PHP scripts of associated with refactorings that have been implemented in the tool AKAAR.
- **css/** : It contains the CSS scripts for styling web pages.
- **php/** : It contains some other PHP scripts like script for uploading a Javascript file. It also contains the parsed XML output obtained by parsing Javascript file using jParser.
- **images/** : It contain images displayed on web pages.
- **uploads/** : It contains the files that have been uploaded.
- **javascript/** : It contains the Javascript scripts for AKAAR.
- **config/** : It contain configuration files for tool AKAAR.

- **parser/** : It contains the lexical analyzer and syntactic analyzer for parsing Javascript code i.e. *jTokenizer* and *jParser.*

- **main.html** : The main page file for tool AKAAR.

So I have tried to keep AKAAR extensible and modifiable.[28]

4.3 Parsing Javascript

4.3.1 Introduction

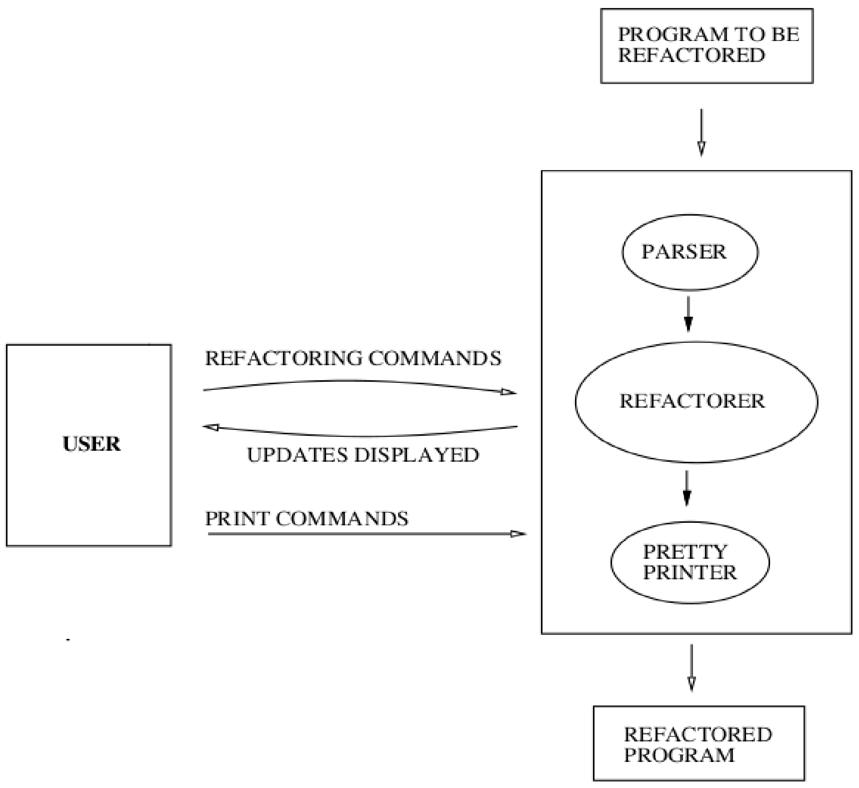


Fig. 4.2 Refactoring process in a refactoring tool

The above figure clearly demonstrates what a key role a parser plays while development of a tool for automation of refactoring techniques. Parsing can be defined as:[42]

"Parsing is the process of analyzing an input sequence in order to determine its grammatical structure with respect to a given formal grammar."

Formally it is also called *Syntax Analysis*.

4.3.2 The Process

The process of parsing can be broadly divided in two steps:

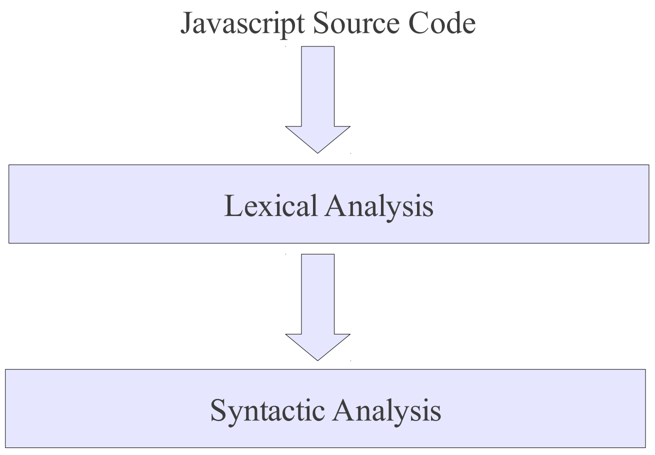


Fig. 4.3 Parsing process

Step 1. Lexical analysis:

The input character stream is split into meaningful symbols (tokens) defined by a grammar of regular expressions.

Example: The lexical analyzer takes "12*(3+4)^2" and splits it into the tokens 12, *, (, 3, +, 4,), ^ and 2.

Step 2. Syntax analysis:

Checking if the tokens form an legal expression, w.r.t. a CF grammar.

Limitations - cannot check (in a programming language): types or proper declaration of identifiers

In our case the *lexical analysis* is done by *jTokenizer* and *syntactic analysis* is done by *jParser*. The CF grammar used by *jParser* for syntactic analysis is a variant of BNF[41]. The entire tool has been developed in PHP programming language and can be easily embedded with a web application by making some minor modifications. The output of *jParser* in our case is an XML(Extensible Markup Language) file called *"test.xml"* in *php* folder. The entire process of parsing is done at the back end. The user of *AKAAR* will never be able to figure out that his/her Javascript code has actually been parsed by a parser in the background. A small example of the parsed output is shown below.

The input Javascript code file is:

```
var i=1;
var j = 'Abhay';

function tune()
{
      alert('I came from an external script! Ha, Ha, Ha!!!!');
}

function rock()
{
      tune();
}

var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings)
    {
      this.strings = newStrings;
    },

    play: function (chord)
    {
      alert('Playing chord: ' + chord);
    },

    print: function (price, currency)
    {
      alert('This guitar is ' + this.color + ', it has ' +
this.strings.length + ' strings' + ' and it costs ' + price +
currency);
    }
};

//using the object
GUITAR.play('Dm7');
GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
debugWrite('this guitar is: ' + GUITAR.color);
GUITAR.print(850, 'USD');
```

First few lines of the corresponding parsed XML output by *jParser* is:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<J_PROGRAM>
   <J_ELEMENTS>
      <J_STATEMENT>
         <J_VAR_STATEMENT>
            <J_VAR>
               var
            </J_VAR>
            <J_VAR_DECL_LIST>
               <J_VAR_DECL>
                  <J_IDENTIFIER>
                     i
                  </J_IDENTIFIER>
                  <J_INITIALIZER>
                     =
                     <J_NUMERIC_LITERAL>
                        1
                     </J_NUMERIC_LITERAL>
                  </J_INITIALIZER>
               </J_VAR_DECL>
            </J_VAR_DECL_LIST>
            ;
         </J_VAR_STATEMENT>
      </J_STATEMENT>
      <J_STATEMENT>
         <J_VAR_STATEMENT>
            <J_VAR>
               var
            </J_VAR>
            <J_VAR_DECL_LIST>
               <J_VAR_DECL>
                  <J_IDENTIFIER>
                     j
                  </J_IDENTIFIER>
                  <J_INITIALIZER>
                     =
                     <J_STRING_LITERAL>
                        'Abhay'
                     </J_STRING_LITERAL>
                  </J_INITIALIZER>
               </J_VAR_DECL>
            </J_VAR_DECL_LIST>
            ;
         </J_VAR_STATEMENT>
      </J_STATEMENT>
      <J_FUNC_DECL>
         <J_FUNCTION>
            function
         </J_FUNCTION>
         <J_IDENTIFIER>
            tune
         </J_IDENTIFIER>
         (
         )
         {
         <J_FUNC_BODY>
            <J_ELEMENTS>
               <J_STATEMENT>
                  <J_EXPR_STATEMENT>
                     <J_EXPR>
                        <J_CALL_EXPR>
                           <J_IDENTIFIER>
```

```
                              alert
                           </J_IDENTIFIER>
                           <J_ARGS>
                              (
                              <J_ARG_LIST>
                                 <J_STRING_LITERAL>
                                    'I came from an external script!
Ha, Ha, Ha!!!!'
                                 </J_STRING_LITERAL>
                              </J_ARG_LIST>
                              )
                           </J_ARGS>
                        </J_CALL_EXPR>
                     </J_EXPR>
                     ;
                  </J_EXPR_STATEMENT>
               </J_STATEMENT>
            </J_ELEMENTS>
         </J_FUNC_BODY>
         }
      </J_FUNC_DECL>
      <J_FUNC_DECL>
         <J_FUNCTION>
            function
         </J_FUNCTION>
         <J_IDENTIFIER>
            rock
         </J_IDENTIFIER>
         (
         )
         {
         <J_FUNC_BODY>

      . . . . . .
      . . . . . .
      . . . . . .
</J_PROGRAM>
```

Hence we can see that this XML output actually demonstrates the overall structure of the Javascript program that we parsed. I will be using this parsed XML output in automating refactoring techniques like *Rename Method*.

4.4 Embedding jParser with AKAAR

jParser is in itself a Javascript parser written in PHP[7] programming language. As no documentation regarding this tool is available from the developer end so it was too difficult for me to figure out how it actually works. But still I tried to figure out and eventually was able to embed it with my refactoring tool with following lines of code in the file *uploadAndAnalysis.php* :

```
require '../jparser/httpdocs/jparser-libs/jparser.php';

      ob_start();

      function generateXML($srcFile)
      {
            $source = file_get_contents($srcFile);
            try
            {
                  $Prog = JParser::parse_string( $source );
            }
            catch( ParseError $Ex )
            {
                  $error      =      $Ex->getMessage()."\n----\n".$Ex-
>snip( $source );
            }
            catch( Exception $Ex )
            {
                  $error = $Ex->getMessage();
            }

            if (!isset($error))
            {
                  header('Content-type:   text/xml;   charset=utf-8',
true );
                  echo '<?xml version="1.0" encoding="utf-8"?>';
                  echo "\n";
                  $Prog->dump( new JLex );
                  $buffer= ob_get_contents();
                  ob_end_clean();
                  $ob_file = fopen('test.xml','w');
                  fwrite ($ob_file, $buffer);
                  fclose ($ob_file);
                  /*echo $buffer;*/
                  /*exit 0;*/
            }
            else
            {
                  echo htmlentities($error,ENT_COMPAT,'UTF-8');
            }
      }
```

ob_start() function will turn output buffering on. While output buffering is active no output is sent from the script (other than headers), instead the output is stored in an internal buffer. The function *ob_get_contents()* is used to copy contents of this internal buffer into a string variable *$buffer.* The line *fwrite ($ob_file, $buffer);* actually writes the extracted contents of input buffer in string variable *$buffer* to the file object *$ob_file.*

4.5 Implementing Rename Method

This refactoring can be defined as:

"If the name of a method does not reveal its purpose then change the name of the method"[1]

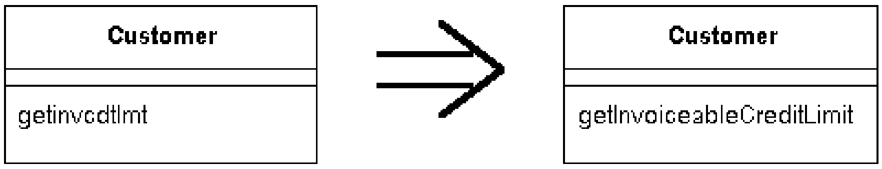


Fig. 4.4 Rename Method

This refactoring technique is one of the most difficult to implement. The reason being that it is not the same as the *Find and Replace* feature available in many of the modern day text editors in which we merely find a word in the given text and replace all occurrences of that word with another word.

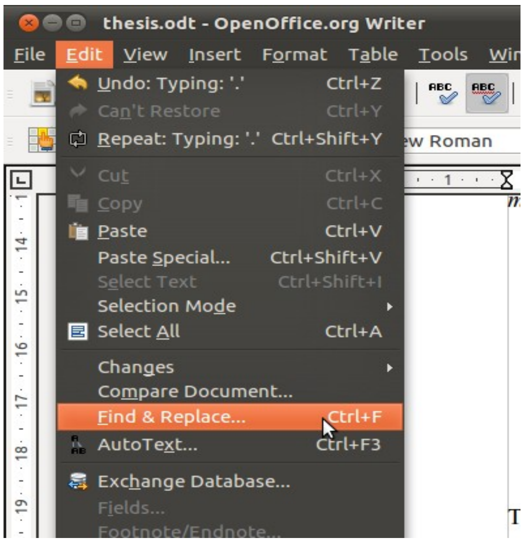


Fig. 4.5 Find and Replace option in a text editor

For example if we have used the *Find and Replace* feature of a text editor like *gEdit* etc. for replacing the method name "tune" with "show_alert" in the following code:

```
var i=1;
var j = 'Abhay';

function tune()
{
      alert('I came from an external script! Ha, Ha, Ha!!!!');
}

function rock()
{
      tune();
}

var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings)
    {
      this.strings = newStrings;
    },

    play: function (chord)
    {
      alert('Playing chord: ' + chord);
    },

    print: function (price, currency)
    {
      alert('This guitar is ' + this.color + ', it has ' +
this.strings.length + ' strings' + ' and it costs ' + price +
currency);
    }
};

//using the object
GUITAR.play('Dm7');
GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
debugWrite('this guitar is: ' + GUITAR.color);
GUITAR.print(850, 'USD');
```

then the editor will replace all the methods named as *tune* in the above code. So the standalone function as well as the function associated with the object GUITAR that have been named as *tune* will be getting renamed as strings and the resultant code will look something like this

```
var i=1;
var j = 'Abhay';

function show_alert()
{
      alert('I came from an external script! Ha, Ha, Ha!!!!');
}
```

```
function rock()
{
      show_alert();
}

var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    show_alert: function (newStrings)
    {
      this.strings = newStrings;
    },

    play: function (chord)
    {
      alert('Playing chord: ' + chord);
    },

    print: function (price, currency)
    {
      alert('This guitar is ' + this.color + ', it has ' +
this.strings.length + ' strings' + ' and it costs ' + price +
currency);
    }
};

//using the object
GUITAR.play('Dm7');
GUITAR.show_alert( ['D', 'A', 'D', 'G', 'B', 'e' ] );
debugWrite('this guitar is: ' + GUITAR.color);
GUITAR.print(850, 'USD');
```

So we can see that not only the standalone function *tune* got renamed but also the *tune* function associated with object GUITAR got renamed. But our intention was not that we just want to rename the standalone function *tune*. In other words we expect the refactoring tool to differentiate methods with similar names in different objects. So in the next few paragraphs I am going to explain the procedure that I have followed to implement this refactoring technique in *AKAAR*. The technique currently works till 1st level of hierarchy but still it can differentiate between standalone methods and methods associated with an object.

Two files *"/aakar/refactorings/r16.php"* and *"/aakar/php/r16.php"* perform the trick for this refactoring in AKAAR. There are two hidden text box elements in /aakar/*refactorings/r16.php* that holds the values for selected name and the name by which to replace the selected name. Ids of these two hidden elements are *"selectedName "* and *"replaceName"*. The file */aakar/php/r16.php* gets the value of these elements using POST method. The reason behind choosing POST is that it allows data to be hidden. The manipulation of whether the selected name by programmer performing refactoring on his/her code belongs to a function associated with an object or a standalone function is done by using tags from the XML output of jParser. The two tags that have been used for this purpose are

- <J_VAR_STATEMENT>
- </J_VAR_STATEMENT>

The concept is simple to understand that if the number of <J_VAR_STATEMENT> is equal to </J_VAR_STATEMENT> before the selected name of function appears then the function is a standalone function because a function associated with an object will not have this case. This is the key concept that is allowing AKAAR to differentiate between standalone functions and function associated with an object because in case of function associated with an object the number of opening and closing tags will not be equal. Note that I am assuming that programmer is declaring objects using literal notation as shown below.

```
var GUITAR = {

    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings) {
     this.strings = newStrings;
    },

    play: function (chord) {
     alert('Playing chord: ' + chord);
    },
    print: function (price, currency) {
     alert('This guitar is ' +
      this.color +
      ', it has ' + this.strings.length + ' strings' +
      ' and it costs ' + price + currency);
    }
   }
```

4.6 Implementing Add Parameter

Add parameter can be defined as:

"A method needs more information from its caller. Add parameter for an object that can pass this information."[1]

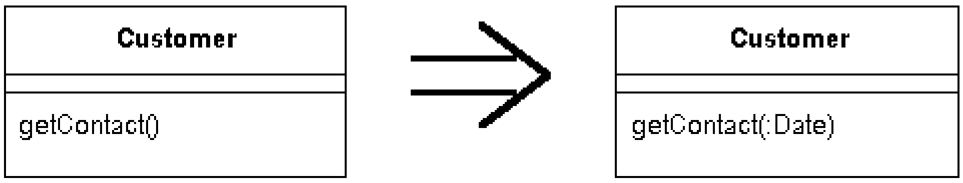


Fig. 4.6 Add Parameter

This is a kind of refactoring that can't be fully automated because adding a parameter and its corresponding statements in a function is something that can only be done by human intuition.

Most of the organizations have got some sort of coding standards that need to be followed while developing a software. These cosing standards help in code maintenance and retain understandability of code in a long term. Keeping these facts in mind I think that it is always better to retain the coding standards set up by an organization. Now most of the coding standards permit only a limited number of parameters in the parameter list of a function because a long parameter list may reduce understandability of code. In order to solve this issue I have tried to follow an approach that can help a programmer applying this refactoring technique to his/her code without neglecting the coding standards.

Whenever a programmer will add a new parameter to the parameter list in function definition he/she is required to select the modified list. Now a small Javascript code in *r1.php* file in *refactorings* directory of my tool AKAAR is going to compute the number of parameters in the list. The first line of configuration file *tool.conf* in *config* directory of my tool is holding the maximum number of parameters permitted by the coding standards. If the number of parameters in parameter list is greater than the one mentioned in *tool.conf* the tool is not going to permit the programmer to add a parameter.

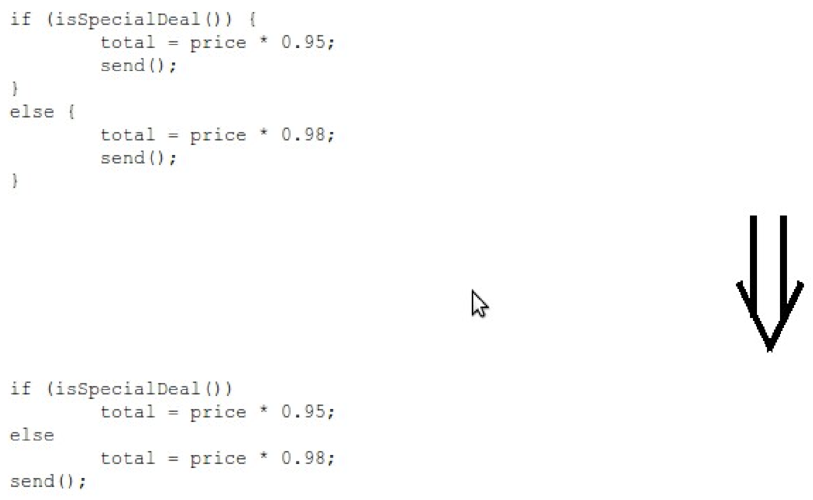
The following code actually performs the trick.

```
var parameters = sel.split(",");
if(parameters.length > parseInt("<?php $fp =
fopen("../config/tool.conf","r"); echo trim(fgets($fp));
fclose($fp);?>"))
            {
            alert("You seem to be violating coding standards... Try
to use parameter object instead...");


location.href="http://www.farzighumakkad.com/aakar/refactorings/r1.ph
p";
            }
            else
            {
                document.getElementById("saveButton").style.visibilit
y = "visible";
                alert("Click save button to save your
modifications...");
            }
```

4.7 Implementing Consolidate Duplicate Conditional Fragments

The refactoring can be defined as[1]:

"The same fragment of code is in all branches of a conditional expression. Move it outside of the expression."



```
if (isSpecialDeal()) {
        total = price * 0.95;
        send();
}
else {
        total = price * 0.98;
        send();
}
```

```
if (isSpecialDeal())
        total = price * 0.95;
else
        total = price * 0.98;
send();
```

Fig. 4.7 Consolidate Duplicate Fragments

This was the toughest of refactorings to automate. Though while reading about this refactoring I never thought that it will turn out to be so complex during its implementation in AKAAR. But finally I have been able to figure out a strategy to implement it.

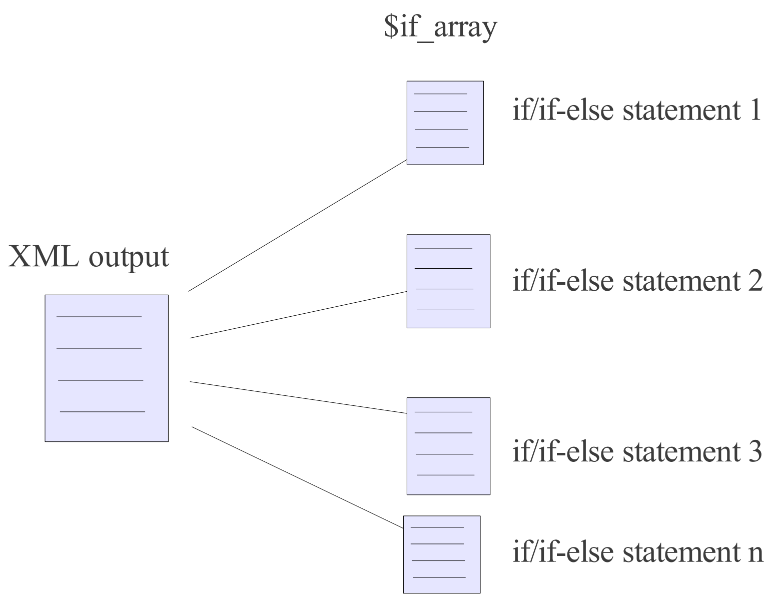


Fig 4.8 Extract if/if-else statements from parsed XML output

So the first step in my approach for implementing this refactoring is that I have extracted *if/if-else statements* in an array *$if_array* by using the XML output of *jParser*. This is possible because if/if-else statements in source code will lie between XML tags <*J_IF_STATEMENT*> and <*/J_IF_STATEMENT*> of XML output of jParser. Now we check each of the extracted if/if-else statement one by one. Now for each case we search for an else statement associated with the if/if-else statement we are looking at the moment. Now a case can occur when a if/if-else statement may contain an if-else statement. Now in this case things become too complex. So in order to sort out the issue I have used a smart approach that I am first checking for an else statement in the current if/if-else statement using <*J_ELSE*> occurrences in the extracted statement. And after an occurrence is detected then I am checking for that whether number of <*J_STATEMENT*> tags are equal to <*/J_STATEMENT*> tags before the <*J_ELSE*> tag appeared. If number of <*J_STATEMENT*> tags are equal to <*/J_STATEMENT*> tags before <*J_ELSE*> in the extracted portion then the detected <*J_ELSE*> is definitely associated with the if statement. In other words we are looking at a valid if-else statement. In the next step I have divided these if-else statements into separate if and else parts in arrays *$if_part* and *$else_part.*

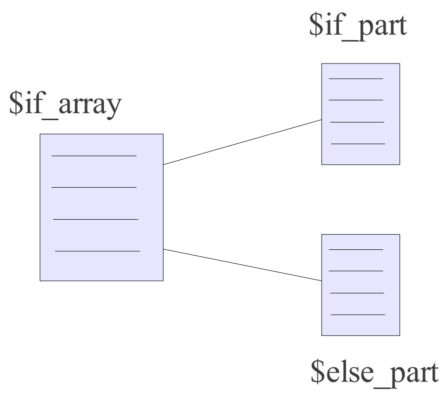


Fig 4.9 Divide each if-else statement in if and else parts

For simple if statements without an else part the values of *$if_part* and *$else_part* has been initialized to *"NULL"*. Next I have extracted the common parts in if and else parts in a three-dimensional array called *$common_part* in my tool. Next I have generated the common portion in if and else parts in *$common_string.* After all this stuff finally I have written the content in our source file in quite a tricky fashion. I hope that the reader will be able to understand things in a much better way if he/she takes a close look at the code of /akaar/refactorings/r3.php because each and everything can't be discussed here.

4.8 Implementing Mention Parameters

As far as *AKAAR* is concerned I have developed a small fraction of this refactoring that detects whether this refactoring needs to be applied on the source code in hand. The trick that I have used to detect this fact is that *r34.php* is detecting whether there is an identifier (i.e. something between <*J_IDENTIFIER*> and <*/J_IDENTIFIER*> in the parsed XML output of the input file) named *arguments* in the input source code. Once my tool detect such an identifier in the XML output of *jParser* it starts finding the function name associated with the *arguments* identifier. It is done by searching element between <*J_IDENTIFIER*> and <*/J_IDENTIFIER*> after the last

<J_FUNC_DECL> before the *arguments* identifier.

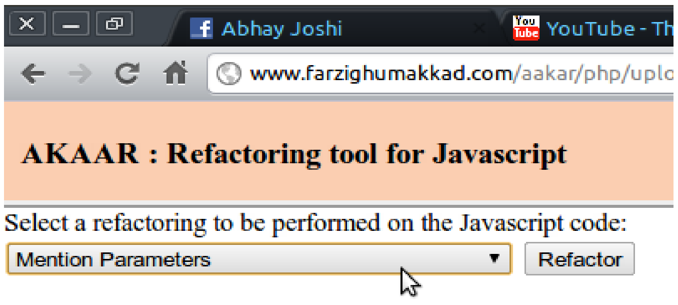
Once AKAAR is able to detect such a function in which there is a need to mention parameters it will automatically display an alert box mentioning names of such functions.

Chapter 5

Results and Discussions

5.1 Mention Parameters

Well to use this refactoring first of all a user of AKAAR need to select this refactoring from the drop-down list after uploading the input Javascript code file.



AKAAR : Refactoring tool for Javascript

Select a refactoring to be performed on the Javascript code:

Fig. 5.1

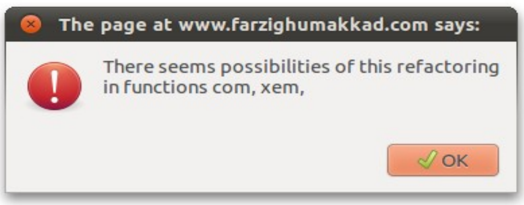
Once we have selected the required refactoring then the tool starts searching for the identifier *arguments* in source code using parsed XML output of *jParser*. The procedure has been described in the previous chapter. I took three inputs to test the working of this refactoring in my tool. The respective outputs are shown just after the code.

Input 1:

```
function com()
{
      var x = 0;

      if(arguments[0] > 0)
      {
            if(arguments[1] > 2)
                  x++;
            x = x + 2;
```

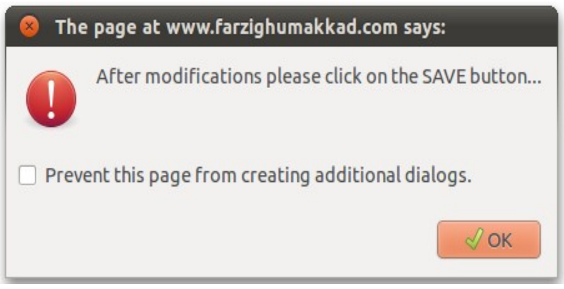
```
        }
        else
        {
                if(arguments[1] > 2)
                        x--;
                x = x - 2;
        }
}

function xyz()
{
        document.write("Hello");
}

function xem()
{
        document.write(arguments[0]);
}
```

Output 1:


The page at www.farzighumakkad.com says:

There seems possibilities of this refactoring in functions com, xem,

Fig. 5.2


The page at www.farzighumakkad.com says:

After modifications please click on the SAVE button...

Prevent this page from creating additional dialogs.

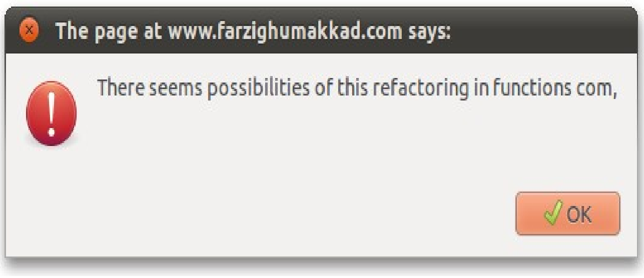
Fig. 5.3

Hence we can see that AKAAR was easily able to detect *arguments* identifier in functions *com* and *xem* and displayed the expected alert boxes.

Input 2:

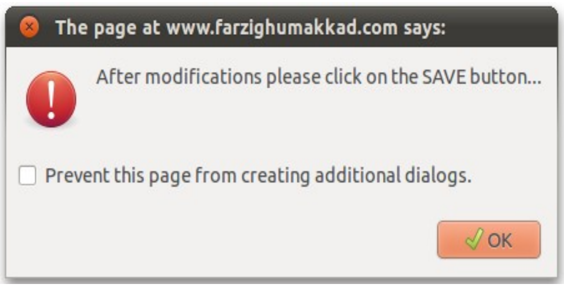
```
function com()
{
        var x = 0;

        if(arguments[0] > 0)
        {
              if(arguments[1] > 2)
                    x++;
              x = x + 2;
        }
        else
        {
              if(arguments[1] > 2)
                    x--;
              x = x - 2;
        }
}

function xyz()
{
        document.write("Hello");
}

function xem(pro_arguments)
{
        document.write(pro_arguments[0]);
}
```

Output 2:


The page at www.farzighumakkad.com says:

There seems possibilities of this refactoring in functions com,

Fig. 5.4


The page at www.farzighumakkad.com says:

After modifications please click on the SAVE button...

Prevent this page from creating additional dialogs.

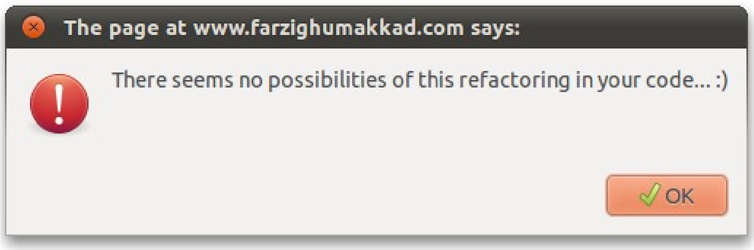
Fig. 5.5

Here we can see that the output is as expected. AKAAR has successfully been able to differentiate between identifiers *arguments* and *pro_arguments*. That is why it is showing a possibility of applying the *"Mention Parameters"* refactoring in function *com* only.

Input 3:

```
function com(p,q)
{
      var x = 0;

      if(p > 0)
      {
            if(q > 2)
                  x++;
            x = x + 2;
      }
      else
      {
            if(q > 2)
                  x--;
            x = x - 2;
      }
}

function xyz()
{
      document.write("Hello");
}

function xem(pro_arguments)
{
      document.write(pro_arguments[0]);
}
```

Output 3:



The page at www.farzighumakkad.com says:

There seems no possibilities of this refactoring in your code... :)

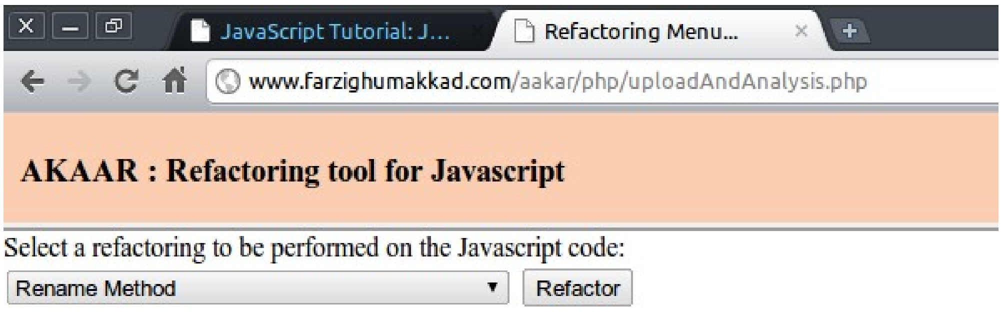
Fig. 5.6

We can see in the above output that as there is no *arguments* identifier found in the parsed output of above code hence we got the appropriate output.

There is still a small limitation of my approach in solving this problem that if an object is defined inside a function with a function that has the arguments identifier associated with it then my technique may produce unexpected names in the list of function names in the alert box. I hope to sort out this issue in the future versions of my tool.

5.2 Rename Method

In order to perform this refactoring first select the method name in the drop down list of the file *UploadAndAnalysis.php* and press the button *Refactor*.

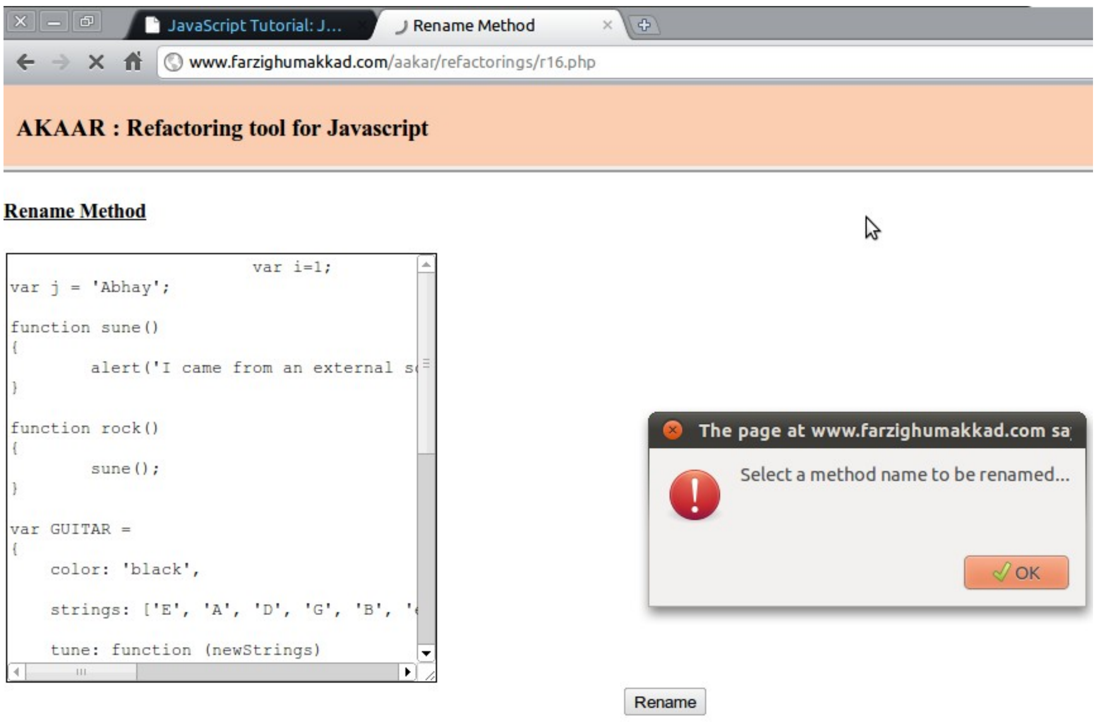

JavaScript Tutorial: J...
Refactoring Menu...
www.farzighumakkad.com/aakar/php/uploadAndAnalysis.php

AKAAR : Refactoring tool for Javascript

Select a refactoring to be performed on the Javascript code:

Fig. 5.7

Once we click the button the next screen shows an alert box asking us to select a method name to refactor. In the same screen we can also find a text area displaying the code in our Javascript file that we have uploaded. A snapshot of the screen is shown below:



JavaScript Tutorial: J... Rename Method

www.farzighumakkad.com/aakar/refactorings/r16.php

AKAAR : Refactoring tool for Javascript

Rename Method

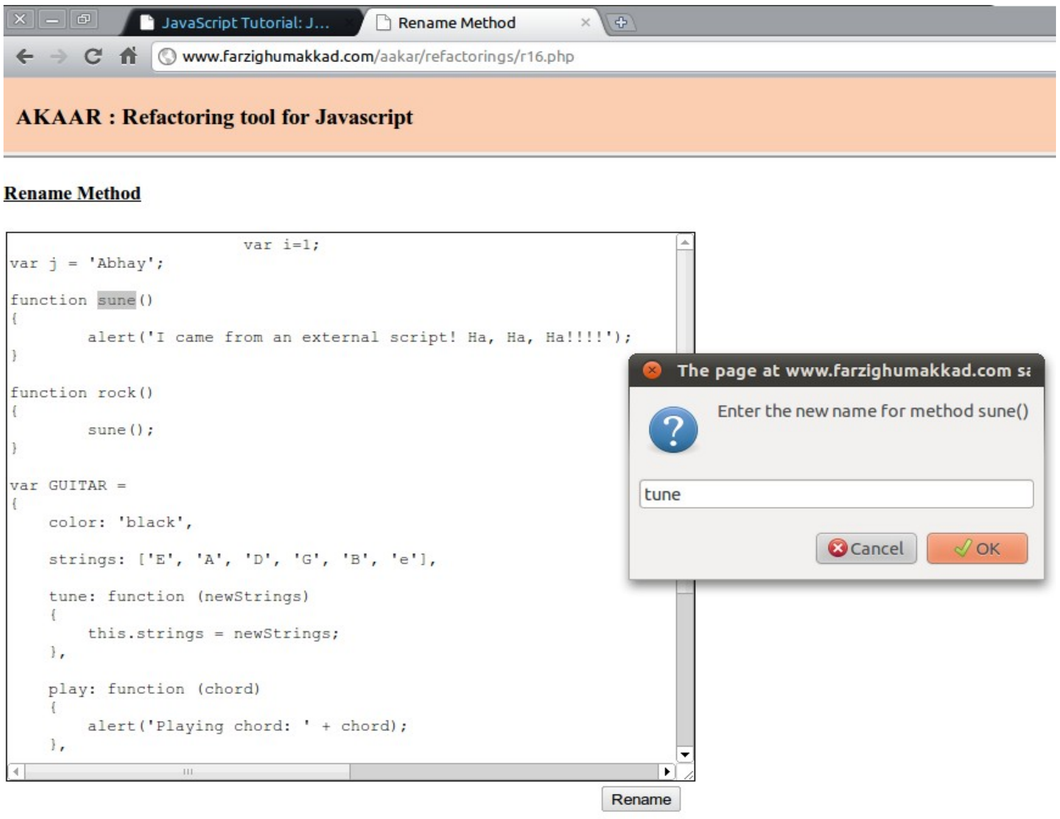
```
                          var i=1;
var j = 'Abhay';

function sune()
{
        alert('I came from an external s
}

function rock()
{
        sune();
}

var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', '

    tune: function (newStrings)
```

The page at www.farzighumakkad.com sa

Select a method name to be renamed...

Fig. 5.8

Once we select a method name to be renamed a prompt box will appear asking the user about the new name of the method. Just fill the name new of the method. As we know that refactoring is a process meant for professional programmers so I am assuming that the programmer using my tool will select only method names during that have been defined in the file. In case of selecting any other thing in the code the renaming process may lead to chaos.



www.farzighumakkad.com/aakar/refactorings/r16.php

AKAAR : Refactoring tool for Javascript

Rename Method

```
                        var i=1;
var j = 'Abhay';

function sune()
{
        alert('I came from an external script! Ha, Ha, Ha!!!!');
}

function rock()
{
        sune();
}

var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings)
    {
        this.strings = newStrings;
    },

    play: function (chord)
    {
        alert('Playing chord: ' + chord);
    },
```

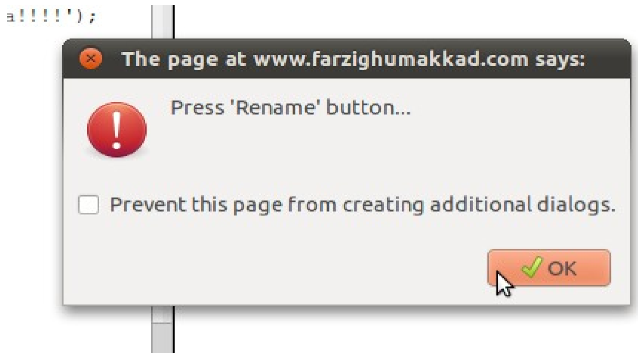
The page at www.farzighumakkad.com s

Enter the new name for method sune()

tune

Fig. 5.9

Once we press *OK* button again an alert box appears that asks the user to press the *Rename* button at the bottom.



The page at www.farzighumakkad.com says:

Press 'Rename' button...

Prevent this page from creating additional dialogs.

Fig. 5.10

Once we press the *Rename* button the uploaded file gets modified and the selected function and all its function calls in the program are renamed.

Example: Suppose the source code of the file that we uploaded was as following:

```
var i=1;
var j = 'Abhay';

function tune()
{
      alert('I came from an external script! Ha, Ha, Ha!!!!');
}

function rock()
{
      tune();
}

var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings)
    {
      this.strings = newStrings;
    },

    play: function (chord)
    {
      alert('Playing chord: ' + chord);
    },

    print: function (price, currency)
    {
      alert('This  guitar  is  '  +  this.color  +  ',  it  has  '  +
this.strings.length  +  '  strings'  +  '  and  it  costs  '  +  price  +
currency);
    }
};

//using the object
GUITAR.play('Dm7');
GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
debugWrite('this guitar is: ' + GUITAR.color);
GUITAR.print(850, 'USD');
```

and we chose the method named *tune* in *line 4* of the above script to be replaced by name *sound*. Then the output will be following:

```
var i=1;
var j = 'Abhay';

function sound()
{
      alert('I came from an external script! Ha, Ha, Ha!!!!');
}

function rock()
{
      sound();
}
```

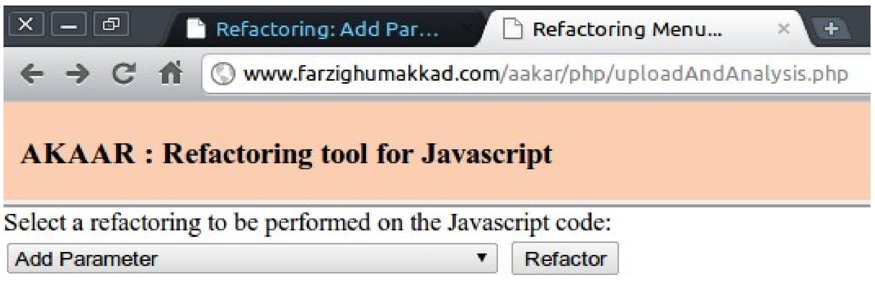
```
var GUITAR =
{
    color: 'black',

    strings: ['E', 'A', 'D', 'G', 'B', 'e'],

    tune: function (newStrings)
    {
      this.strings = newStrings;
    },

    play: function (chord)
    {
      alert('Playing chord: ' + chord);
    },

    print: function (price, currency)
    {
      alert('This  guitar  is  '  +  this.color  +  ',  it  has  '  +
this.strings.length  +  '  strings'  +  '  and  it  costs  '  +  price  +
currency);
    }
};

//using the object
GUITAR.play('Dm7');
GUITAR.tune( ['D', 'A', 'D', 'G', 'B', 'e' ] );
debugWrite('this guitar is: ' + GUITAR.color);
GUITAR.print(850, 'USD');
```

So we can see that though there is another method with the name *tune* that has been declared in the object *GUITAR* but my tool has been able to successfully detect that the selected method is a standalone function. Hence the function *tune* associated with object GUITAR remains unaltered.

The interesting feature of my approach is that I have tried to design the refactoring in such a manner that I am trying to free the user from entering the context in which the function has been declared. In other words my tool will automatically detect whether the method is associated with an object or whether the method is a standalone function with no object associated with it. I have been successful in doing this at a level 1 hierarchy. Still I need to figure out how to deal with methods in case of a multi-hierarchical situations.

5.3 Add Parameter

In order to implement this refactoring first we need to choose this refactoring from drop down list. After selecting it press *Refactor*.



Refactoring: Add Par...

Refactoring Menu...

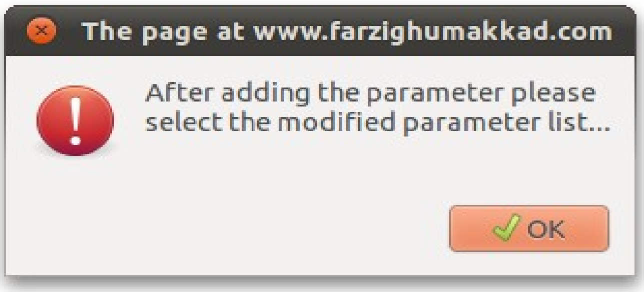
www.farzighumakkad.com/aakar/php/uploadAndAnalysis.php

AKAAR : Refactoring tool for Javascript

Select a refactoring to be performed on the Javascript code:

Fig. 5.11

In the next screen an alert box will pop up on the screen as shown below asking the programmer to select the modified parameter list once all the necessary modifications have been done.



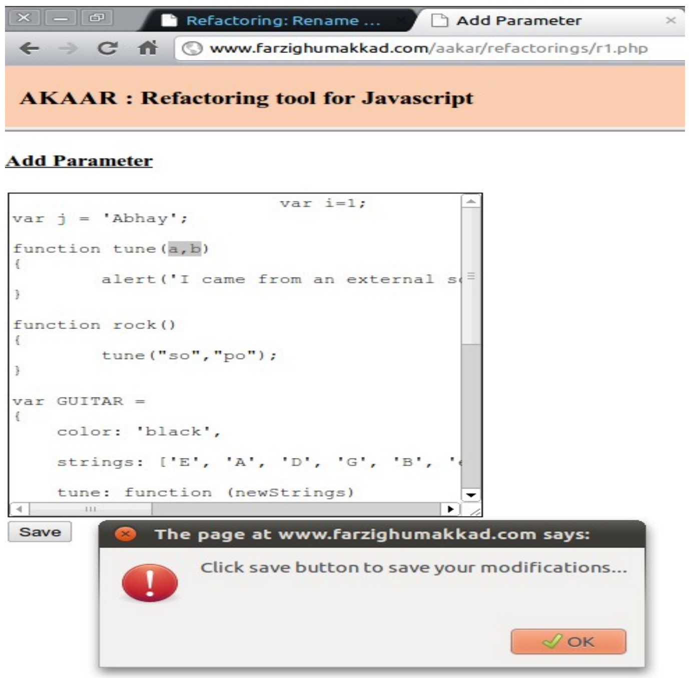
The page at www.farzighumakkad.com

After adding the parameter please select the modified parameter list...

Fig. 5.12

Now after adding parameters select the modified parameter list. By default the maximum number of permissible parameters have been set to *2* in *tool.conf*. On selection either of the following two screens are going to happen

Case 1: The number of parameters do not exceed the limit set in *tool.conf.*



AKAAR : Refactoring tool for Javascript

Add Parameter

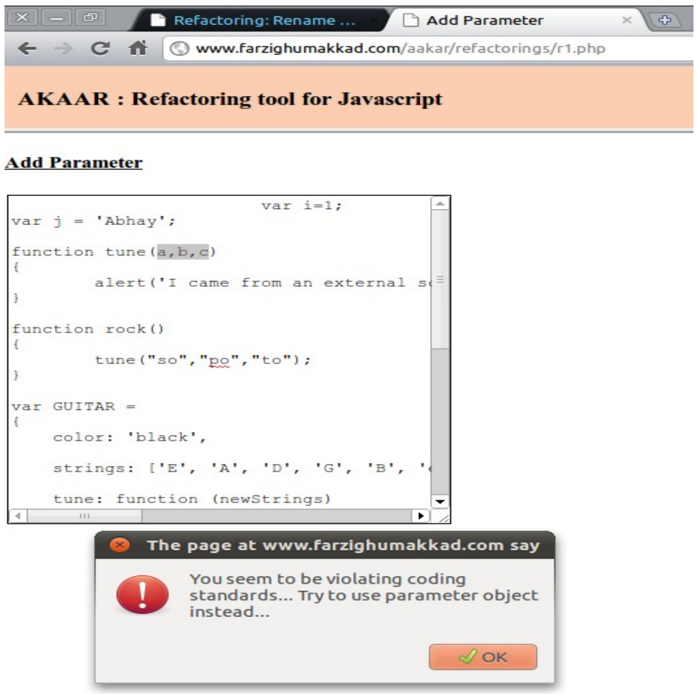
```
                                 var i=1;
var j = 'Abhay';
function tune(a,b)
{
          alert('I came from an external s
}
function rock()
{
          tune("so","po");
}
var GUITAR =
{
     color: 'black',
     strings: ['E', 'A', 'D', 'G', 'B', '
     tune: function (newStrings)
```

The page at www.farzighumakkad.com says:

Click save button to save your modifications...

Fig. 5.13

Case 2 : The number of parameters exceed the limit set in *tool.conf.*



AKAAR : Refactoring tool for Javascript

Add Parameter

```
                                 var i=1;
var j = 'Abhay';
function tune(a,b,c)
{
          alert('I came from an external s
}
function rock()
{
          tune("so","po","to");
}
var GUITAR =
{
     color: 'black',
     strings: ['E', 'A', 'D', 'G', 'B', '
     tune: function (newStrings)
```

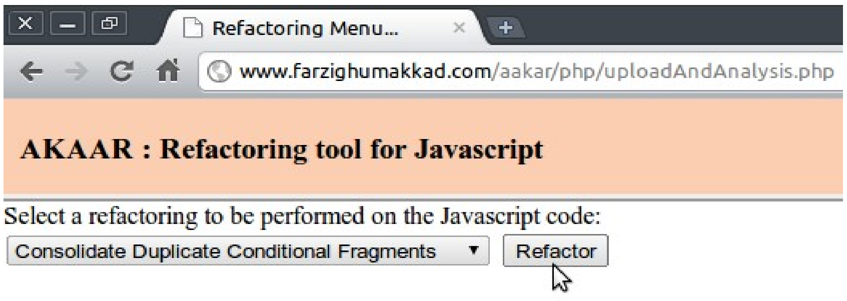
The page at www.farzighumakkad.com say

You seem to be violating coding standards... Try to use parameter object instead...

Fig. 5.14

5.4 Consolidate Duplicate Conditional Fragments

This refactoring is really the one that has been completely automated. In order to use this simply select this refactoring to be applied to the code from the refactoring menu and click the *Refactor* button.


Refactoring Menu...
www.farzighumakkad.com/aakar/php/uploadAndAnalysis.php

AKAAR : Refactoring tool for Javascript

Select a refactoring to be performed on the Javascript code:

Consolidate Duplicate Conditional Fragments
Refactor

Fig. 5.15

Once we have clicked the button the duplicate fragments in conditional statements automatically get consolidated. Here is the list of input files that I used for checking my results and their corresponding outputs:

Input 1:

```
function com(x)
{
      if(x > 0)
      {
            document.write("Hello");
            x++;
      }
      else
      {
            document.write("Hello");
            x--;
      }
}
```

Output 1:

```
function com(x,c)
{


if(x>0){
      x++;
}
else
{
      x--;
}
document.write("Hello");

}
```

Input 2:

```
function com(x,c)
{
      if(x > 0)
      {
            if(c > 2)
                  c++;
            x++;
      }
      else
      {
            if(c > 2)
                  c++;
            x--;
      }
}
```

Output 2:

```
function com(x,c)
{


if(x>0){
      x++;
}
else
{
      x--;
}


if(c>2)c++;

}
```

So we can see that this refactoring technique can easily consolidate duplicate fragments in conditional statements. The only issue with my approach is that it is still not able to deal with situations in which *if* and *else* parts have exactly same statements. Moreover there are certain formatting issues with duplicate fragments in if-else statements that need to be dealt in future versions of AKAAR.

During my work of implementing this refactoring I observed a very unusual case that I will like to mention in this part of this thesis. Suppose we give the following input to the refactoring tool:

```
function com(x,c)
{
      if(x>0)
      {
            if(c > 2)
                  c++;
            x++;
            document.write(c);
      }
      else
      {
            if(c > 2)
                  c++;
            x--;
            document.write(x);
      }
}
```

Then the output on applying *"Consolidate Duplicate Conditional Fragments"* is as follows:

```
function com(x,c)
{
      if(x>0)
      {
            x++;
            document.write(c);
      }
      else
      {
            x--;
            document.write(x);
      }
      if(c > 2)
                  c++;
}
```

But we can see that the expression

```
if(c > 2)
      c++;
```

is playing an important role in the very next statement i.e. *document.write(c);* in the *if* part of *if-else* statement. So removing this duplicate fragment can lead to unexpected output too. This fact has not been discussed in Martin Fowler's classic[1]. Hence I believe that it will prove to be of some use to the reader of this thesis.

Chapter 6

Conclusions

6.1 Contributions

The major contributions of this thesis and project work are as following:

1. As far as code refactoring is concerned till now most of the work has been done for class based object-oriented languages like C++ and Java but in this thesis work I have tried to take the initiative of refactoring prototype based object-oriented languages.

2. Every task has got some pros and cons associated with it and same is the case with refactoring. This thesis also discusses several pros and cons of refactoring that have not been discussed in books available on the topic of refactoring.

3. This thesis discusses the advantages of refactoring a given code for reusability purpose rather than rewriting code from scratch for a given problem.

4. Programming languages can have language specific refactoring techniques associated with them. While I was developing a web application “Pikia” for my minor project I faced an understandability problem on using with() statement of Javascript in my project's source code. So I have proposed a new language specific refactoring technique called *“Removing with statement”* on the basis of a blog article by Douglas Crockford [21] for Javascript in this thesis.

5. Most of the people associated with object-oriented development think that object-oriented languages is all about classes and their instances. They think that OO features of inheritance, encapsulation etc. can be implemented only by using the concept of classes and their instances. This thesis describes the difference between class and prototypes based OO languages i.e. the two approaches of implementing object-oriented concepts of inheritance, encapsulation etc. in a programming language.[9]

6. Today the term refactoring is used in various contexts. This thesis discusses how the domain of refactoring has evolved in the past two decades and its applications at various levels.

7. I have tried to implement some refactoring techniques from the catalog[2] that can be applied in case of Javascript in my refactoring tool *AKAAR[60]*. I have also discussed important details regarding implementation of these techniques in chapter 4 of this thesis. I hope that this initiative taken by me will be able to bring good results in near future.

8. All the web developers know that Javascript is going to play a key role in the next generation HTML5 web applications.[24] The refactoring technique proposed by me, the modified refactorings and the syntactic constraints in case of refactorings in Javascript will definitely help developers in developing automated refactoring tools for Javascript.

9. Very few people are able to understand how certain aspects in a programming language work. Same is the case with Javascript. In fact I believe that Javascript is much more complex syntactically when compared to other object-oriented languages like Java and C++. I have mentioned some syntactic aspects of Javascript like the way Javascript deals with arguments that has led me to propose a new refactoring called *"Mention Parameters"* for Javascript in this thesis.

10. Very few people know about the parsers available for Javascript. In this thesis I have reused an open-sourced and freely available parser called *Jparser[40]*. I believe that the code of my tool *AKAAR[60]* will help readers in understanding how to embed a *Jparser* in your web application.

11. The project work covered in this thesis clearly demonstrates the power of open-source philosophy. It was only because of the code of *jParser* that was shared by Tim Whitlock on the Internet that I was able to develop *AKAAR*. If he had not shared his code then I would never have been able to embed a Javascript parser in my tool.

12. I believe that with this thesis I have been successful in demonstrating the fact

that a lots of new refactorings can be found for Javascript due to some of its unique characteristics. I hope that readers of this thesis will definitely like to explore this topic further.

13. Earlier I thought that as refactoring is all about dealing with patterns in code which are called as code smells so I will be able to deal with them merely by using regular expressions. But I was wrong because without using a parser with the tool I don't believe now that it is possible to develop a tool that can automate refactorings. Hence this thesis demonstrates what's the important role played by a parser in refactoring tools.

6.2 Limitations of Approach

I won't say that this thesis work is a masterpiece in any sense because there are several limitations in my approach by which I performed my research. Following are the limitations that I will like to mention about my approach to solve the problem:

- The renaming being done by *Rename Method* refactoring is unable to perform renaming of a method at any level of hierarchy. It can only rename standalone methods and methods associated with an object but directly used by the object.
- The renaming being done by *Rename Method* refactoring is not able to categorize the name of a method in a string literal.
- The tool developed my me is a standalone application. I need to embed it in some IDE or framework.
- Still many refactorings haven't been discussed by me in the thesis. Moreover many refactorings have been simply avoided by me in this thesis for the sake of avoiding their complex implementation in case of Javascript.
- Many refactoring techniques have still not been discussed by me due to time constraints.
- Many refactoring techniques being implemented in case of class-based object-

oriented languages like Java and C++ need to be modified so that they can be adapted for the case of Javascript.

- The format of the output code in case of *"Consolidate Duplicate Conditional Fragments"* isn't too good as far as readability of code is concerned.

6.3 Other Areas of Future Research

What I have developed in a span of six months is nothing but just a platform above which I can develop a lot of things in the near future. In the next few months I will like to work on the following areas to enhance the capabilities of my refactoring tool *AKAAR.* Moreover I will also like to find new refactoring techniques for code written in Javascript.

- I need to work on a technique for renaming methods at a hierarchical level.
- Testing plays a critical role I havn't performed testing of my tool AKAAR yet so I need to do that in the near future.
- Automatic detection of code clones in a given program can help developers in reducing redundancy in their codes. Moreover reduction of code clones can also improve the performance of software. I need to work on code clone analysis[12] to automatically find code clones in the given Javascript source code.
- Manual detection of code smells in a given program is a hectic task. There are several lightweight techniques proposed by researchers[20][21] that can be applied in case of Javascript with slight modifications. So I need to embed automatic detection of code smells in my refactoring tool AKAAR.
- Still a lot of work needs to be done for finding syntactic constraints related to Javascript refactoring techniques.
- The domains of refactoring and testing are closely associated with each other. Sometimes refactoring may introduce some bugs in our code. The best way to

avoid such circumstances is to perform testing simultaneously with refactoring. In future research work I will try to embed unit testing tools for Javascript like *jsUnit, Jasmine* etc. in my tool.

- Right now my tool *AKAAR* is just a standalone application that I have developed merely to understand the implementation of refactoring techniques. But I believe that its not so useful unless it's embedded in an web development IDE or a framework. So in future I will like to embed my tool to a framework or an IDE.
- I will also like to find refactoring possibilities in code written for Javascript using libraries like *prototype.js*
- Still lots of refactorings applicable to class-based object-oriented languages need to be adapted to the prototype-based languages like Javascript.
- The next version of AKAAR will have the proposed refactoring *"Removing with()"* automated.

6.4 Summary

Javascript is already playing a major role in modern days web applications and I hope that in the next generation of HTML5 web applications Javascript is going to play a key role. So we can't neglect Javascript merely as a client side scripting language. Javascript is going to play a vital role in the next decade. Thousands of lines of code will be written in Javascript for these applications. Hence we can't neglect refactoring of this OO code written in Javascript as it will be reused in many other similar applications too. Moreover being a prototype based object-oriented language its syntactic and semantic constraints are a bit different than class based object-oriented languages like Java and C++. The amount of work I have done during this thesis work is just a mere initiative for refactoring Javascript code. A lot needs to be done in this field as can be seen in the long list of future research areas in the previous section. Refactoring is a hard task to perform but when a developer gets used to it it becomes a part of his/her programming practice. At last I will like to end this thesis by mentioning that refactoring can be made an efficient and less time consuming task by developing advance tools that can perform automatic detection of refactorings and assist developer while applying them to the source code. I hope that this thesis will motivate readers in contributing more to this interesting topic.

REFERENCES & BIBLIOGRAPHY

1. **Martin Fowler *et al.***, *Refactoring: Improving the Design of Existing Code,* 1st ed., Addison-Wesley Professional, 1999.

2. **Martin Fowler**, *Alpha List of Refactorings*[Online], Available: http://www.refactoring.com/catalog/index.html

3. **M. Fowler**, *Refactoring*, in Proc. of the 24th Int. Conf. on Software Eng., 2002, pp. 701, DOI 10.1109/ICSE.2002.146344

4. **ECMA (2009)**, *Standard ECMA-262* [Online], Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

5. **David Flanagan**, *JavaScript: The Definitive Guide*, 5th ed., O'Reilly, 2006.

6. **Jeffrey E.F. Friedl**, *Mastering Regular Expressions*, 3rd ed., O'Reilly, 2006.

7. **Kevin Tatroe *et al.***, *Programming PHP*, 2nd ed., O'Reilly Media, 2006.

8. **R.S. Pressman**, *Software Engineering: a Practitioner's Approach*, 6th ed., McGraw-Hill, 2005.

9. **Kent Beck**, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2001.

10. **Antero Taivalsaari**, *Classes vs. Prototypes: Some philosophical and historical observations,* Nokia Research Center, Helsinki, Finland, 1996.

11. **William F. Opdyke**, *Refactoring Object-Oriented Frameworks,* Ph.D. Thesis, Dept. Comp. Sci., Univ. Of Illinois, Urbana, 1992.

12. **Yoshiki Higo *et al.***, *ARIES: Refactoring Support Environment based on Code Clone Analysis*, in 8th IASTED Int. Conf. on Software Eng. and Applications, Cambridge, MA, USA, 2004.

13. **Ling Lan *et al.***, *A Middleware-based Approach to Model Refactoring at Runtime*, in 14th Asia-Pacific Software Eng. Conf., pp. 246-253, 2007, DOI 10.1109/ASPEC.2007.45

14. **D. Manolescu *et al.***, *Volta: Developing Distributed Applications by Recompiling,* in IEEE J. Software, vol. 25, issue 5, pp. 53-59, 2008, DOI 10.1109/MS.2008.131

15. **D. Dig**, *A Refactoring Approach to Parallelism,* in IEEE J. Software, vol. 28, issue 1, pp. 17-22, 2011, DOI: 10.1109/MS.2011.1

16. **Jason Hayes *et al.***, *Extreme programming: a university team design*

experience, in Canadian Conf. on Electrical and Computer Eng., vol 2, pp. 816-820, 2000*,* DOI 10.1109/CCECE.2000.849579

17. **Gustavo Rossi *et al.***, *Refactoring For Usability In Web Applications,* IEEE J. Software, vol. PP*,* issue 99, 2010, to be published. DOI 10.1109/MS.2010.114
18. **Q.D. Soetens and S. Demeyer** , *Studying the effect of refactorings: a complexity metrics perspective* , in 7th Int. Conf. on the quality of Information and Communications Technology, pp. 313-318, 2010, DOI 10.1109/QUATIC.2010.58
19. **Yi Wang**, *What motivate software engineers to refactor source code? evidences from professional developers*, in IEEE Int. Conf. on Software Maintenance, pp. 413-416, 2009, DOI 10.1109/ICSM.2009.5306290
20. **D.C. Atkinson and T. King**, *Lightweight Detection of Program Refactorings*, in Proc. of the 12th Asia-Pacific Software Eng. Conf., 2005, DOI: 10.1109/APSEC.2005.76
21. **Y. Tsuda *et al.***, *Detecting Occurrences of refactoring with Heuristic Search*, in 15th Asia-Pacific Software Eng. Conf., pp. 453-460, 2008, DOI: 10.1109/APSEC.2008.9
22. **Programming Tools Group**, *Refactoring bugs in Eclipse* [Online]. Available: http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports/eclipse
23. **Douglas Crockford** (2006), *with() statement considered harmful* [Online]. Available: http://www.yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/
24. **Christina Warren** (2011), *How Javascript and HTML5 are remaking the web* [Online], Available: http://mashable.com/2011/03/17/javascript-html5/
25. **B. Drapsin *et al.***, *An Approach to Parallelization of Legacy Software*, in 1st IEEE Eastern European Conf. on the Eng. of Computer Based Systems, pp. 42-48, 2009, DOI 10.1109/ECBS-EERC.2009.8
26. **JsLint** [Online], Available: http://www.jslint.com/
27. **T. Sigmund *et al.***, *Mining Software Evolution to Predict Refactoring*, in 1st Int. Symp. on Empirical Software Eng. and Measurement, pp. 354-363, 2007, DOI: 10.1109/ESEM.2007.9
28. **S. Yamamoto and K. Maruyama**, *Design and implementation of an extensible and modifiable refactoring tool*, in 13th Int. Workshop on Program Comprehension, pp. 195-204, 2005, DOI: 10.1109/WPC.2005.17
29. **JsHint** [Online], Available: http://jshint.com/jshint.js
30. **G. Rossi *et al.***, *Model Refactoring in Web Applications*, in 9th IEEE Int. Workshop on Web Site Evolution, pp. 89-96, 2007, DOI: 10.1109/WSE.2007.4380249

31. **T. Kamiya et al.**, *CCFinder: a multilinguistic token-based code clone detection system for large scale source code*, in Software Engg. IEEE Transactions, pp. 654-670, July 2002

32. **M.V. Mäntylä**, *Empirical software evolvability - code smells and human evaluations*, in IEEE Int. Conf. on Software Maintenance, pp. 1-6, 2010, DOI: 10.1109/ICSM.2010.5609545

33. **J.M. Bieman *et al.***, *The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction*, in Int. Conf. on Software Testing Verification and Validation, pp 141-150, 2009, DOI: 10.1109/ICST.2009.21

34. **R. Conradi *et al.***, *Maintenance and agile development: Challenges, opportunities and future directions*, in IEEE Int. Conf. on Software Maintenance, pp. 487-490, 2009, DOI: 10.1109/ICSM.2009.5306278

35. **M. Schäfer and O. Moor**, *Of Gnats and Dragons: Sources of complexity in implementing refactorings*, in Proceedings of the 3rd workshop on refactoring tools, ACM, 2009.

36. **Doug Wallace *et al.***, *Extreme Programming for Web Projects*, 1st ed., Addison-Wesley Professional, 2002.

37. **Harold Elliotte Rusty**, *Refactoring HTML*, 1st ed., Pearson Education India, 2008.

38. **Wikipedia, *Javascript*** [Online], Available: http://en.wikipedia.org/wiki/JavaScript

39. **Cecily Barnes**, *More Programmers going "Extreme"*[Online], Available: http://news.cnet.com/2100-1040-255167.html

40. **Tim Whitlock**, *jParser and jTokenizer released*[Online], Available: http://timwhitlock.info/blog/tag/jparser/

41. **Tim Whitlock**, *Grammar for jParser*[Online], Available: http://timwhitlock.info/wp-content/uploads/2009/03/jas.bnf

42. **Wikipedia, *Parsing***[Online], Available: http://en.wikipedia.org/wiki/Parsing

43. **Gmail** [Online], Available: http://mail.google.com/mail/

44. **Facebook** [Online], Available: http://www.facebook.com/

45. **YouTube** [Online], Available: http://www.youtube.com

46. **Picnik** [Online], Available: http://www.picnik.com

47. **Google Analytics** [Online], Available: http://www.google.com/analytics/

48. **Blogger** [Online], Available: http://www.blogger.com

49. **Google Docs** [Online], Available: http://docs.google.com

50. **Google Books** [Online], Available: http://books.google.com

51. **World Wide Web Consortium**, *HTML5 Specification* [Online], Available: http://dev.w3.org/html5/spec/Overview.html

52. **Extreme Programming** [Online], Available: http://www.extremeprogramming.org/

53. **Visual Studio 2010** [Online], Available: http://www.microsoft.com/visualstudio/en-us

54. **Eclipse** [Online], Available: http://www.eclipse.org

55. **NetBeans** [Online], Available: http://www.netbeans.org

56. **Firefox web browser** [Online], Available: http://www.mozilla.com/en-US/firefox/new/

57. **Internet Explorer web browser** [Online], Available: http://windows.microsoft.com/en-IN/internet-explorer/products/ie/home

58. **Opera web browser** [Online], Available: http://www.opera.com

59. **Chromium web browser** [Online], Available: http://www.google.com/chrome

60. **AKAAR refactoring tool** [Online], Available: http://www.farzighumakkad.com/aakar/main.html

61. **Fabrice Bellard's x86 emulator in Javascript** [Online], Available: http://bellard.org/jslinux/

62. **Microsoft Office 2010** [Online], Available: http://office.microsoft.com/en-in/

63. **OpenOffice** [Online], Available: http://www.openoffice.org/

64. **w3schools**, *HTML Tutorial* [Online], Available: http://www.w3schools.com/html/default.asp

65. **w3schools**, CSS *Tutorial* [Online], Available: http://www.w3schools.com/css/default.asp

66. **Rasmus Lerdorf *et al.***, *Programming PHP,* 2nd ed., O'Reilly Media, 2006

67. **Glen Ford**, *Programming to the Interface In JavaScript: yes, it can be done... er, I mean faked* [Online], Available: http://knol.google.com/k/programming-to-the-interface-in-javascript-yes-it-can-be-done-er-i-mean-faked#

68. **Robert Husted and J.J. Kuslich**, *Server-side JavaScript: developing integrated Web applications*, Addison Wesley, 1999

69. **Tom Mens and Tom Tourwe** , *A survey of software refactoring*, in IEEE trans. on Software Engg., Vol. 30, No. 2, pp.126-139, 2004, DOI 10.1109/TSE.2004.1265817

APPENDIX A : Regular Expressions

In computing, a regular expression[6], also referred to as *regex* or *regexp*, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

A Regular Expression is the term used to describe a codified method of searching *invented*, or *defined*, by the American mathematician Stephen Kleene.

Extended Regular Expressions (EREs) were defined in IEEE POSIX 1003.2 (Section 2.8).EREs are now commonly supported by Apache, PERL, PHP4+, Javascript 1.3+, MS Visual Studio, MS Frontpage, most visual editors, vi, emac, the GNU family of tools (including grep, awk and sed) as well as many others.

I am not going to run too deep into explaining the working of Regular Expressions in this section of this thesis. Hence I will like to briefly describe some of the important metacharacters that I have used in my project.

Metacharacter	Meaning
[]	Match anything inside the square brackets for ONE character position once and only once, for example, [12] means match the target to 1 and if that does not match then match the target to 2 while [0123456789] means match to any character in the range 0 to 9.
-	The – (dash) **inside square brackets** is the 'range separator' and allows us to define a range, in our example above of [0123456789] we could rewrite it as [0-9]. We can define more than one range inside a list, for example, [0-9A-C] means check for 0 to 9 and A to C (but not a to c). **NOTE:** To test for - inside brackets (as a **literal**) it must come first or last, that is, [-0-9] will test for - and 0 to 9.
^	The ^ (circumflex or caret) **inside square brackets** negates the expression (we will see an alternate use for the circumflex/caret **outside** square brackets later), for example, [^Ff] means anything except upper or lower case F and [^a-z]

	means everything except lower case a to z. **NOTE:** Spaces, or in this case the lack of them, between ranges are very important.
^	The ^ (circumflex or caret) **outside square brackets** means look only at the beginning of the target string, for example, **^Win**will not find Windows in **STRING1** but **^Moz** will find **Moz**illa.
$	The $ (dollar) means look only at the end of the target string, for example, fox$ will find a match in 'silver **fox**' since it appears at the end of the string but not in 'the fox jumped over the moon'.
.	The . (period) means any character(s) in this position, for example, **ton.** will find **tons**, **tone** and **tonneau** but not **wanton** because it has no following character.
?	The ? (question mark) matches the preceding character 0 or 1 times only, for example, colou?r will find both color (0 times) and colour (1 time).
*	The * (asterisk or star) matches the preceding character 0 or more times, for example, tre* will find tree (2 times) and tread (1 time) and trough (0 times).
+	The + (plus) matches the previous character 1 or more times, for example, tre+ will find tree (2 times) and tread (1 time) but not trough (0 times).
{n}	Matches the preceding character, or character range, n times exactly, for example, to find a local phone number we could use [0-9]{3}-[0-9]{4} which would find any number of the form 123-4567. Note: The **-** (dash) in this case, because it is outside the square brackets, is a literal. Value is enclosed in braces (curly brackets).
{n,m}	Matches the preceding character at least n times but not more than m times, for example, 'ba{2,3}b' will find 'baab' and 'baaab' but NOT 'bab' or 'baaaab'. Values are enclosed in braces (curly brackets).
()	The ((open parenthesis) and) (close parenthesis) may be used to group (or bind) parts of our search expression together.
|	The | (vertical bar or pipe) is called **alternation** when technically speaking and means find the left hand OR right values, for example, gr(a|e)y will find 'gray' or 'grey'.

Table A.1 Terminology for Regular Expressions