2020

Refactoring an existing code base to improve modularity and quality

Souradeep Bhowmik
Iowa State University

Refactoring an existing code base to improve modularity and quality

by

Souradeep Bhowmik

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Simanta Mitra, Co-major Professor
Gurpur Prabhu, Co-major Professor
Ying Cai

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

This thesis work is dedicated to my parents, Baran Kumar Bhowmik and Soma Bhowmik, and my brother Barnadeep Bhowmik, who mean the world to me. Their constant words of encouragement and support throughout my life has been the reason for my success. They have been with me through highs and lows and have always given me the guidance required to help me be successful in life.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

NOMENCLATURE

ADL	Application Description Language
JS	JavaScript
NPM	Node Package Manager
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
POJO	Plain Old Java Object
UI	User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
JWT	JSON Web Token

ACKNOWLEDGMENTS

I would like to take this opportunity to express my heartfelt gratitude to all who have supported me throughout this journey and have extended a helping hand in shaping my academic success. I would also like to extend my thanks to my Program of Study committee for their exceptional guidance and support throughout the course of this thesis. This thesis would be incomplete without the constant feedback from Dr. Simanta Mitra and the guidance from Dr. Gurpur Prabhu. I would also like to thank my committee member Dr. Ying Cai for his support and encouragement throughout this journey.

I also take this opportunity to express my gratitude for the unbounded support and love from my family, although no amount of words will ever be enough to describe it.

In addition, I would also like to thank my friends, colleagues, the department faculty, and staff for making my time at Iowa State University a wonderful experience and the journey towards earning my degree a memorable one.

ABSTRACT

Code written in modern programming languages (such as Java) can be almost impossible to understand and maintain due to poor design and coding practices used during its development. Instead of redeveloping the entire code from scratch (which is an expensive and time-consuming proposition), typically a series of refactoring steps are applied to make the software better in terms of both design and coding quality, which translates to better user experience because the maintainability and scalability of the application is increased. In this project we consider an existing code base that was written hastily in Java and was really poor in terms of design and code quality. We share our experiences in refactoring this code base in order to make it modular and with improved design and code quality. We first analyzed the existing code base to identify areas for improvement and then used certain benchmark metrics to guide the refactoring. We present a comparison of the final state of the code with the original code base to demonstrate the use of good software development practices.

CHAPTER 1. INTRODUCTION

The notion of software engineering existed even before the emergence of modern programming languages and writing a good quality code has become the topic of massive research over the years as the discipline grew and saw a boom in the late 20th century. However, the *"quality"* of software still appears to be a vague term for many. It is important to understand that the quality of software does not depend on just one factor, it is the collection of a multitude of metrics that help to verify the quality of code written. The development of software relies on the design and a poorly designed software system leads to bad quality of code. Over time, the entire software development lifecycle has evolved to adapt to the ever changing discipline. However, even after all the metrics and guidelines available at the disposal of software engineers, the code sometimes does not reflect good standards of development. We take a look at one such code base, Application Description Language [1], that has poor quality of code and share our experiences of refactoring it from a developer's perspective in terms of changes that a developer has to make to refactor the code base.

Refactoring code as an idea has been around for a long time. Even before the term was coined, it was being used actively as part of the software engineering process. Arguably, the most useful part of refactoring is its contribution towards maintainability. Software systems, over time, will inevitably require changes to be implemented, but the motivation behind the changes can vary. For example, a change may be required for addition of a new feature, a bug fix, improve readability of the code, upgrading the version of technologies used etc. A good quality of code helps reduce resources required for change, but the state of code may not always reflect good quality. Refactoring the code base in such a scenario helps to maintain the software better

as it promotes good practices. However, one important thing to remember is that the refactoring of a code base should not lose any of the functionalities of the previous version. This is one of the reasons that legacy systems require more allocation of resources to refactor as important business logic often hides deep in the code and it is very tough to identify and transform.

The Application Description Language (ADL) was identified to be one such code base that has a lot of bottlenecks to maintainability in its implementation. ADL is a tool that can be used to generate client-server based applications by providing the program with configurations as input in the form of a JSON (JavaScript Object Notation) file. There are 3 parts to ADL: the parser that parses the input configuration and generates the output, the server code generation template that is used by the parser to generate the server output and, the client code generation template that is used by the parser to generate client output. The ADL in its previous state had a lot of bottlenecks to growth in the form of many poor design and development choices. Also, inheriting this work from the previous developers creates a void in understanding the code because of its complexity which contributes to the already sizeable list of bottlenecks. In this report, we provide an initial analysis of the code base and identify specific portions of code that need refactoring. We have defined 5 qualitative standards of modularity to help guide our analysis of the code, viz. cohesion, coupling, code organization, code reusability and extensibility. This analysis serves to guide our refactoring efforts and we have come up with transformations that will get rid of the problems.

The refactoring is done in phases where each different type of refactoring required is first linked to the portion of code that requires it, which is followed by an implementation of the changes to the code. This work presents all the different types of refactoring that was required and a brief summary of the implementation strategies.

We conclude this work by presenting a comparison of the features available in the previous version of ADL and the refactored version to prove that the refactoring of the code base has successfully retained all of the functionalities. This comparison is done manually where the feature set of previous version of ADL is manually identified and the same is done with the refactored version and is checked for any potential loss of functionalities.

The rest of the report is organized as follows. Chapter 2 talks about the related work in this field. Chapter 3 introduces the standards of software quality that were used to analyze the modularity of the code base, followed by an analysis of the existing code base in Chapter 4. Chapter 5 discusses in detail our refactoring efforts and Chapter 6 provides a comparison and evaluation of our refactored version. The report is concluded in Chapter 7 by presenting the general observations from refactoring and scope of future work.

CHAPTER 2. LITERATURE REVIEW

Refactoring is essentially a two-step process. The first step is to incorporate the transformations to the design and code, and the second step is to verify that there is no loss of functionality during the transformation phase. It originated in Smalltalk circles and quickly made its way into being an essential software engineering process. One of the first prominent works in this field was the refactoring tool for Smalltalk [4]. It defines refactoring as a *"behavior preserving transformation"*. The authors argue that the purpose of refactoring is to essentially make the code be more reusable and easier to understand, rather than introduce more features. More work in this field quickly started being published around the same time and in the following years [7]-[17]. All of the research presented different ways of incorporating better quality of code by use of refactoring.

In recent years, many have presented techniques of automating the refactoring process, including the verification of the functionalities [18]-[20]. In the work presented in [19], the authors argue that legacy systems are much harder to refactor because of different reasons. They make use of model equivalence checking to verify that a legacy system has not lost any functionality during the refactoring process. If there is a loss, counter examples are generated, which can be then used as reference to adapt the implementation and refine the model. [18] and [20] both present an approach to automate the refactoring process. In [18], the authors aim to make a recommender system for guiding the addition of features through refactoring. They have proposed a system that will help the developer add new features with the help of refactoring, as the developer often has to make transformations that end up introducing more code smells, rather than remove them. The authors in [20] propose an alternative to traditional refactoring of legacy

code, which helps them introduce custom refactoring principles. They have taken an iterative approach to making an automated refactoring tool for this purpose.

In most of the previous work done in this field, there seems to be a focus on the design of the code and a lack of focus on the developer's perspective on refactoring. In [5], the authors report their experiences in developing an automated refactoring tool by taking feedback from the developers of 5 different software companies. This is one of the first work to share their experiences from the perspective of the developer. In this paper, the authors have presented experiences in 2 categories. The first is the challenges to automate refactoring transformations, and the second is the perception of the developer about the automatically refactored code. This is different from the work presented in our thesis because we have presented the types of transformations that a developer has to make at the grassroots level to refactor by analyzing a poorly written code base. Our focus is not on automating the transformations but provide an experience report on the tangible changes that a developer has to undertake.

CHAPTER 3. QUALITATIVE REFACTORING STANDARDS

The objective of refactoring a code is to introduce better quality that directly translates to better user experience because the maintainability and scalability of the application is increased. There have been many metrics of software refactoring, both qualitative and quantitative, that were introduced over the years to help guide the process of refactoring, but the end goal still remains the same, a better quality of code. Our goal is to set 5 qualitative standards of modularity viz. cohesion, coupling, code organization, code reusability and extensibility to analyze the existing code and also use them to evaluate the refactored code in order to compare them.

Cohesion

A software system, complex or otherwise, will have multiple parts to it. Implementing these parts in separate code modules is the idea behind modularity in code. It stems from the idea that each module should be responsible for executing only once aspect of the desired functionality. The degree to which each element inside of a module are related is known as cohesion. Using this measure as reference, a developer can transform the code to achieve high cohesion between the elements of a module, which is the desired effect of modularity.

Coupling

Introducing cohesion alone in the software code is not enough to warrant a good quality of code. The reason behind this is that the interaction between these modules play a very important role in determining the complexity and it reflects in the user experience. The degree to which all modules interact with each other is known as coupling. Low coupling of software modules ensures that the interaction and consequently the complexity and readability of software is controlled.

Code organization

A code written in any format, whether it is a single method with thousands of lines of code or a very organized, modular code, is compiled the same way by a machine. It then introduces a very important dilemma for the developer on whether to concentrate on just the output or make it readable for the people as well. The authors in [3] argue that "*programs must be written for people to read, and only incidentally for machines to execute"*. A software is never perfect, it is only made better over time by revisioning and refactoring. Code organization thus plays a very important role towards achieving a good quality of code that is maintainable and this makes it easier for a person to revisit and make changes.

Code reusability

An important aspect of modern software engineering is to write code that is reusable. Functions are a way of incorporating this in many programming languages, such as Java. Reusable pieces of code are not limited to only one application. For example, the Node Package Manager has an online repository of packages that can be imported and used in any project and can be customized as the developer sees fit. A reusable module helps the developer get rid of repeating code, which increases code maintainability and readability.

Extensibility

Extensibility in code promotes future growth. This growth can be in terms of adding a new functionality, a bug fix etc. and it is very important for the modern software development process to create opportunities for growth. It is a direct consequence of modularity in code and refactoring a code base to incorporate modularity will result in the system being extensible.

CHAPTER 4. ANALYSIS OF EXISTING CODE

The ADL, as mentioned earlier, has three parts to its code. The first part deals with the Java based parser that takes input configuration from the user and updates pre-constructed templates using the Mustache compiler library. The other two parts of the ADL deals with the actual code that gets generated for the client application and the server application. We first present an overview of the ADL and then analyze all the three sections of the existing code in rest of the chapter.

Application Description Language

This part is the core of the application, which has the Java based parser that can take a user configuration in the form of a JSON text file and parse it to generate the client server application as its output. This ADL application has 9 class files; one for the main method to start the application, two POJOs to map the input configuration JSON text file as Java objects (one for mapping the entire input and the other one to map only the front end UI configuration), and six files that help with parsing the input and generating its corresponding output. All of these classes are organized under 3 packages, a *"launcher"* package for the main class, *"model"* package for the POJOs and a *"parsing"* package for all the parser code, as seen from Figure 4-1. An introduction to all these individual classes is presented next.

ADLApplication.java

This class holds the main method of the application, which expects two arguments. The first argument is a reference to the input configuration JSON file from the system directory and the second argument is the output directory where the application code gets generated.

Input.java

This is a POJO representing the input fields from the JSON configuration text file.

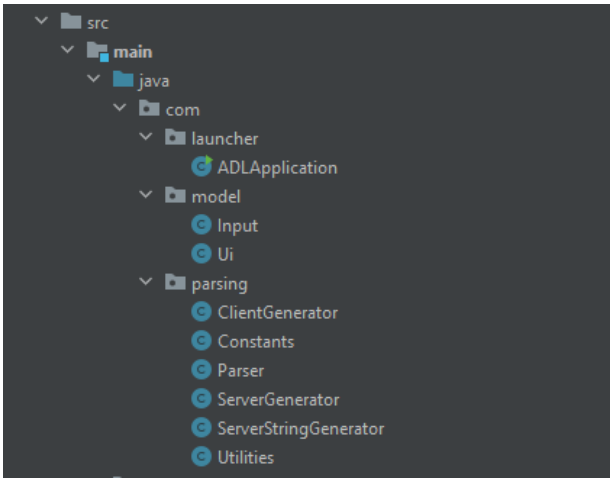

Figure 4-1: Code organization of ADL

Ui.Java

This is a POJO representing the UI section of the input fields.

Parser.java

This class contains the method that takes as input the input file reference and output directory from the main method and calls the corresponding server and client generator methods from the respective classes.

ServerGenerator.java

This class is responsible for generating the server code using the pre constructed templates and the input configuration.

ServerStringGenerator.java

This class has multiple methods that return the extra lines of code that needs to be added to the template files for introducing custom functionalities to the application.

ClientGenerator.java

This class also serves a similar purpose as the server generator class in that it takes the input configuration and updates template files for the client application using Mustache library.

Utilities.java

This class contains three methods for deleting a file, copying a file and removing certain lines of code from a file. For removing lines of code, it makes use of the Constants file to get the position and the corresponding number of lines for deletion from the file.

Constants.java

This file contains very important resources as hard coded values, that are used throughout the ADL application. Figure 4-2 is a snapshot of the class from the previous version of ADL.

```
class Constants {
    static String baseSourcePathServer = "C:\\Users\\Preethi\\Desktop\\webapp-language-master\\src\\main\\resources
    static String baseSourcePathClient = "C:\\Users\\Preethi\\Desktop\\webapp-language-master\\src\\main\\resources

    static int modelLastLine = 22;

    static int controllerSaveLocation = 32;
    static int controllerGetAllLocation = 23;
    static int controllerAddLocation = 23;
    static int controllerTimestampAddLocation = 39;
    static int controllerLoginLocation = 43;

    static int repositoryLastLocation = 10;
    static int repositoryVerifyLocation =9;

    static int modelMapPosition = 33;
    static int modelSaveFormStartPosition = 31;
    static int modelRegisterFormStartPosition = 32;
    static int modelRegisterScriptPosition = 75;
    static int modelButtonStartingPosition = 22;
    static int modelFormStartingPosition = 30;

    static int jsFileBeginningPosition = 2;
    static int jsChatHidePosition = 19;

    static int hyperLinkStartPosition = 18;

    static String fileSaveLocation = "C:\\\\Users\\\\tanma\\\\Downloads\\\\";
}
```

Figure 4-2: Contants.java file contents

The rest of the chapter is dedicated to the analysis of the previous ADL code. The analysis is presented in the form of a table where each of the three parts of ADL are evaluated against the quality standards that were defined earlier. A color coded severity of the problem existing in each of these parts is also presented to help with the refactoring implementations.

Analysis of ADL

Table 4-1 provides an evaluation of the previous version of ADL in terms of its modularity in the 5 quality standards that were introduced earlier. It can be clearly seen from this evaluation that the client template of the ADL requires the most attention because it violates all but one of the quality standards. The server template is fairly well coded but could use some improvements in organization. The ADL parser as a whole also requires a lot of refactoring to comply to the standards set. An explanation of the violations is presented below.

Table 4-1: ADL evaluation

Part of ADL	Cohesion	Coupling	Organization	Reusability	Extensibility
Parser	High	Very tightly coupled	Needs improvement	Needs improvement	Not extensible
Server template	High	Loosely coupled	Needs improvement	Reusable	Extensible
Client template	Low	Very tightly coupled	Organized	Not reusable	Not extensible

Module cohesion

The ADL in itself is fairly well developed in terms of introducing modularity to the code. However, the templates used by ADL and the corresponding generated application code is very poorly developed. The generated client code especially has just one *"control.js"* file that handles all of the interaction for the web page, including fetching data from the server. As an example, a template of this file contains 26 lines of code initially, but the output generated using 5 entities in its input has 565 lines of code, and it will only increase with the number of entities and UI aspects that gets introduced in future updates. The client application code is thus developed with very low cohesion as this one module has various unrelated elements inside of it.

Module coupling

The ADL code is very deeply dependent on the Constants file. This file, as previously mentioned, has hard coded values for various things used for generating the output. The hard coded values represent line numbers in the template files for the generator classes to read and add extra lines of code to. Hard coded values also represent the number of lines that are needed to be deleted from the template files. Any change in the template files will require the developer to spend extensive amounts of time trying to identify and update the specific variable(s) that is (are) affected by the change. This high coupling resulted in our first execution of the inherited code in a failure, which can be seen from the Figure 4-3. The client template itself represents a very tightly coupled code, as everything in the client end is dependent on a single file, as discussed earlier.

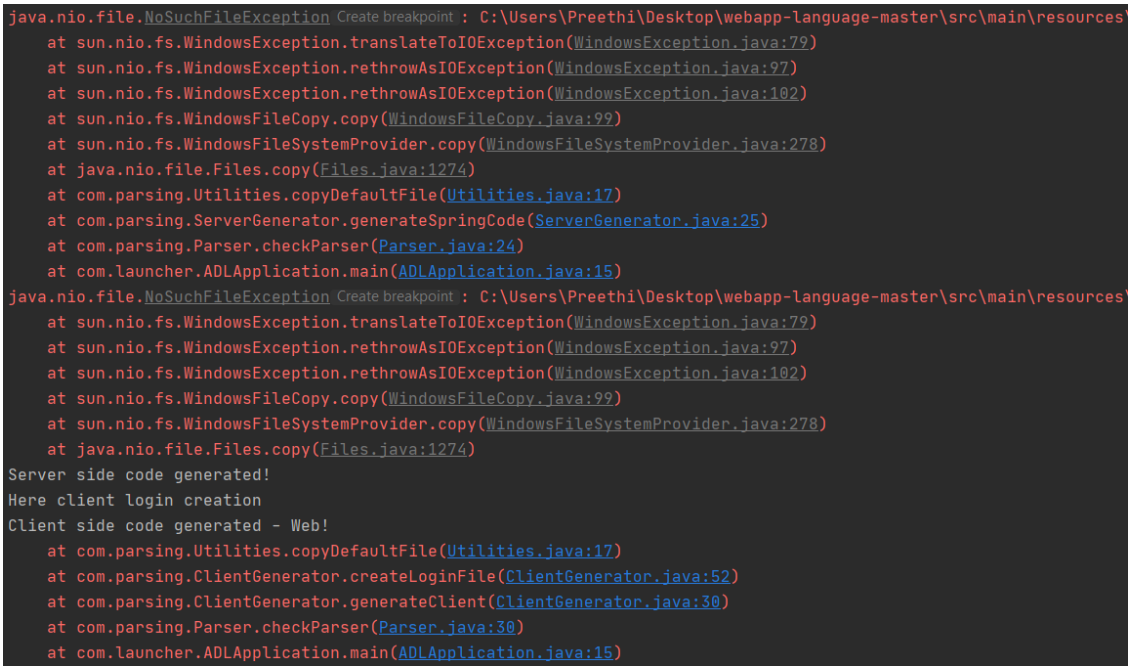

Figure 4-3: Build failure for ADL

Code organization

Looking back at Figure 4-1 we see that the utilities class is packaged under the parser. All other classes in this package help in processing and generating the output application, however this class still exists in this package. This utilities class should be packaged in a different directory, so as to maintain the grouping together of similar things in one place. For the server template, authentication needs to be separated out and organized in a package that represents authentication feature, as it is completely different from where it currently resides in.

Code reusability

The ADL parser makes use of the Mustache compiler to update the pre constructed template files. The lines of code that is required to achieve this is repeated multiple times throughout the implementation and we identified 9 separate instances of repeated code for this purpose. The client template is developed in such a way that every piece of functionality available for interacting with the data corresponding to all the entities in the input has repeated pieces of code. All of this needs refactoring to get rid of repeated code.

Extensibility

The ADL application in its existing implementation does not promote growth. There are too many instances of bad design choices and hasty implementation techniques that exist throughout the application, both in its parser code and also the template for the server and client application code. This leaves it being a developer's worst nightmare for anyone who intends to do future work on ADL. Any new addition of functionalities, especially on the client end, will require major revisit of the template code and development will be sluggish and costly due to the time being spent on understanding the code.

The implementation details of all the different types of refactoring that were required for ADL is discussed in the next chapter. Environment setup details can be found in Appendix.

CHAPTER 5. DESIGN AND IMPLEMENTATION

This chapter presents an in depth discussion on the implementation details of the different types of refactoring that were required for ADL. It is important to note that the current implementation of client end template using JavaScript is extremely tough to extend. Therefore, for refactoring the client end, it was first required that this part be redesigned and developed using a method that makes it easier to maintain and grow. A JavaScript library, React JS, was used for this purpose and the details of implementation are presented later in the chapter.

Types of refactoring done on ADL

Change class design

The two Java classes representing the input configuration were modified in their design to accommodate for more fields in the input. This was done to help the parsing be done better in terms of being able to convey the same information in a better way.

Split variable assignment

This type of refactoring separates the variable declaration and assignment. This is not required for all declarations, but the usage of hard coded values for finding the base source directories of client and server template prompted for a better approach which required fetching these details at the run time. A function was implemented to return the absolute path of these directories from the local file system. It was therefore important to separate the assignment of values to these variables, as the return from the method may raise an exception and the separation allows us to enclose the assignment with proper exception handling.

Repackaging

This type of refactoring means reorganizing the source code into proper packages. For the ADL parser code, the utilities and constants classes were extracted into their own package as

they served more of a helper class role, than directly being used in the parser class for generating the output. The server end code was refactored to extract the authentication from a controller class to its own implementation, which is discussed later in the chapter.

Slide statements

This refactoring means rearranging the lines of code inside of a module. The previous version of ADL had function declarations in between variable declarations inside of multiple classes, and so it was important to rearrange these declarations to group together similar lines of code for better understanding and readability.

Removing of hard coded values

The constants file, as previously discussed, makes it extremely difficult for the developer to make any kind of changes to the ADL. To get rid of the tight dependency of the output generator code with the constants file, comments were used as tags to identify specific line numbers inside of the server and client template files. This enabled us to find the line numbers during program execution and was not static as it was implemented earlier. A method was implemented to help fetch line numbers from the template files using these tags and this ensures that the return always represents the current value. Also, all changes to the template files were implemented to be additions of lines of code, instead of addition and deletion. This helped to eliminate specifying the number of lines to delete from a template, which was also present in the constants file in its previous implementation.

Class usage of static members

The methods inside of the utilities class has a static modifier, meaning that they are a property of the class. The usage of these methods only requires for the class to be imported as they can be used directly with the help of the class name. The previous implementation created

objects to use these methods and this was an unnecessary step, hence all occurrences of usage of the utilities class was modified to change the access using the class name only.

Extract functions

The piece of code that takes the input configuration and updates the template files were extracted to a separate method to promote reusability. The method to get line number of a template file by usage of comment tags was also extracted to a function for a similar purpose. The piece of code that gets the absolute path in the local file system for a directory was extracted and moved to a function as well. All these different refactoring helps the code be more modular and reusable.

Simplify code

There is a library called lombok for Java that lets developers make use of annotations to avoid writing repetitive code, like getters, setters, override of *"toString()"* method etc. We have used this library to simplify the template for the server side. This refactoring ensures that the readability of code is improved and hence it promotes future growth.

Extract modules

Similar to extracting functions, extracting modules means creating separate modules out of existing code so as to increase module cohesion. As discussed earlier, the authentication feature of the generated server application template was implemented inside of a controller class. However, this needs extraction to a module and packaging accordingly because authentication is a separate feature altogether. The authentication feature was introduced using a JWT based authentication to increase security of the generated app. This implementation generated 9 new class files and were packaged separately in a new security package in the template.

Rename variables

This type of refactoring was required throughout the code and helped to increase the readability of the ADL code base.

Meaningful comments

Comments were introduced in various places to make the code more readable for the developer, which would decrease the barrier to understanding of the code implementation for a developer who has not previously worked with the ADL code base.

Remove dead imports and code

The code base was refactored to get rid of dead code which does not contribute to the application anymore. This includes imports that are not used as well.

The client end code required major attention to restructure the implementation. At its previous state, it was extremely difficult to do so. We have thus made use of a JavaScript library, React JS, to introduce better code quality to the client end code. The rest of the chapter presents the design and implementation details of the React JS client end code.

Design Architecture of React app

React JS development enforces certain design principles that helps in maintainability by enabling the developer to make the code modular with the help of components. Figure 5-1 shows the organization of modules and the corresponding packages of the generated application. Inside of the components folder we have 4 separate folders, viz. Auth, Misc, Models and Utils. Auth and Utils are self-explanatory. The Misc folder contains an *"About.jsx"* and a *"Contact.jsx"* component that are responsible for rendering the information specified in the input configuration file. Models folder contains all of the components representing the entities specified in the input.

They all share the same functionalities, but only those functionalities that are specified in the input are allowed in the component, rest all are disabled.

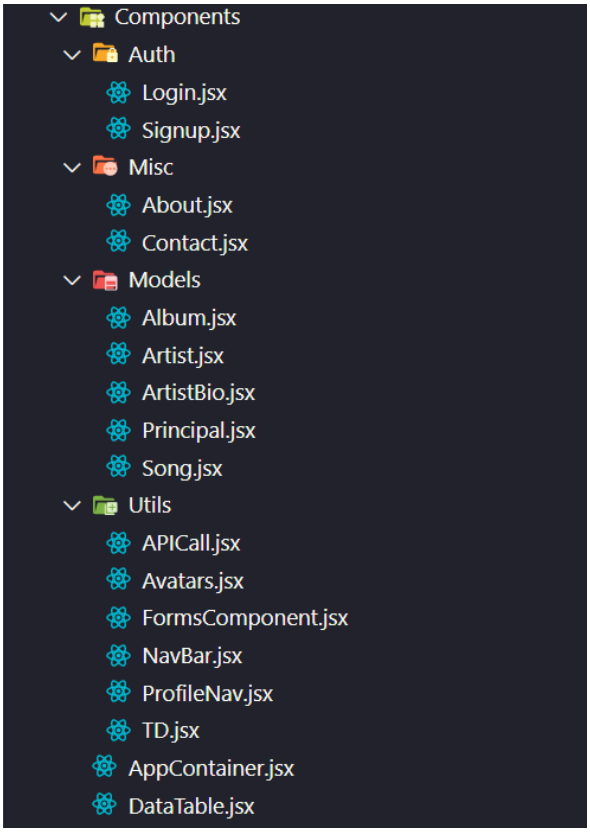

Figure 5-1: Component organization

Figure 5-2 presents a visual representation of the component hierarchy in the generated application using an input with 5 entities, with appropriate legends at the bottom of the figure indicating the folder that they belong to and also the level of the component in the overall hierarchy. At the root of the hierarchy is the *"AppContainer.jsx"* component, followed by the *"Login.jsx"* and *"Signup.jsx"* and so on. At the second level we can see the 5 model components corresponding to the 5 input entities, and specific actions are allowed/disabled inside of these components based on the input configuration.

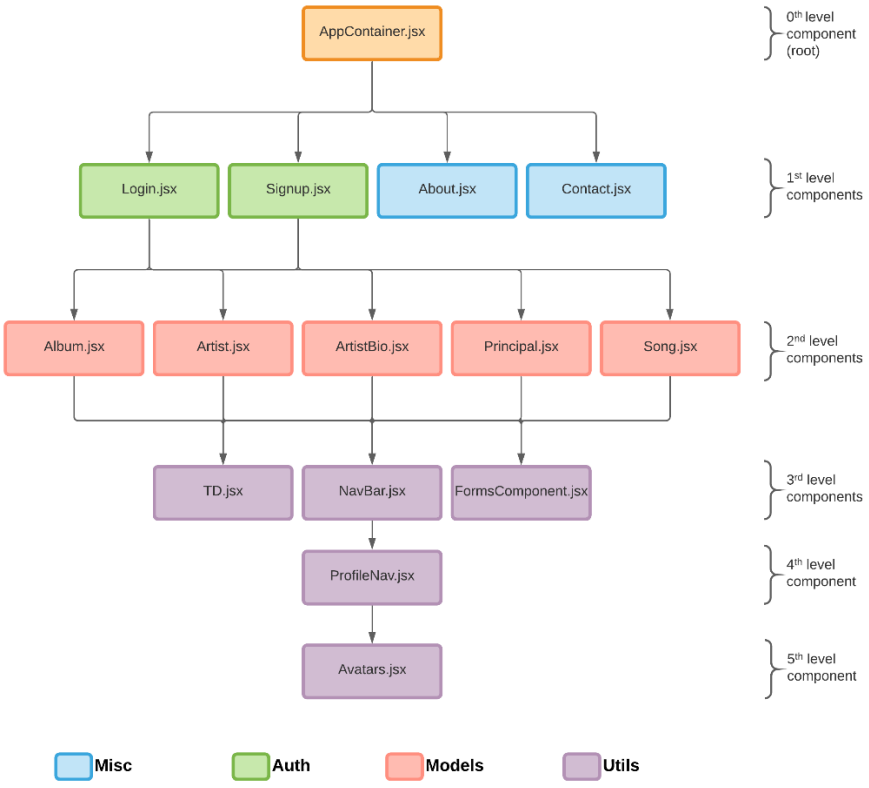

Figure 5-2: React app component hierarchy

Implementation of React app

The client end code required a cleaner, modern, and modular implementation that promotes maintainability, scalability, and reusability. The rest of the chapter is dedicated to describing the implementation details of each component and explaining the design decisions taken in order to achieve our modularity targets.

AppContainer.jsx

This is the root of the application; this is where the application boots from. This component therefore provides us the opportunity to set up the navigation paths to be used in the application. We have used “react-router-dom” package to set up the routes as this is a very easy to use and easy to maintain package.

Login.jsx

This is a stateful component that is rendered to give the user a login interface. The state of the component is updated with the help of default HTML "onChange" property that updates on every key press. A variable is used to keep track of form errors and is updated with corresponding values for errors in input for either name or password from the user. The form submission is not allowed until all error messages are cleared (by entering valid entries into the text input sections). The validity of the password is checked with the help of a regular expression. After a successful login, the component also keeps a copy of the JSON Web Token (JWT) in the local storage of the client, as shown in Figure 5-3, to send with every subsequent server request so that the client does not have to validate again.

Figure 5-3: Client local storage with JWT

Signup.jsx

This component is very similar to the login component in its implementation. Similar to the login component, the validity of the input is checked against a regular expression. Upon successful request submission to the server, this component redirects the user to the login page to now login to the application.

Contact.jsx

This page renders contact information provided in the input configuration.

About.jsx

This component is identical to the contact component. The only difference with the other one is that the text is different as it is a separate field in the input configuration.

All Model Components

All model components have the same basic structure. Figure 5-4 shows an example of how every component looks like. Every component has a navigation bar with links to all model components. All model components make use of the “componentDidMount()” React lifecycle method to check for user login information. In the login component we discussed that the JWT is saved in the client’s local storage and the lifecycle method reads this local storage to check for a JWT. Only if a valid token is found in the storage, the client is able to access the component. Else, the client is redirected to the login screen to work on authentication again. Each of the buttons in a model component represents actions that can be performed with the model’s data which is defined in the input configuration.

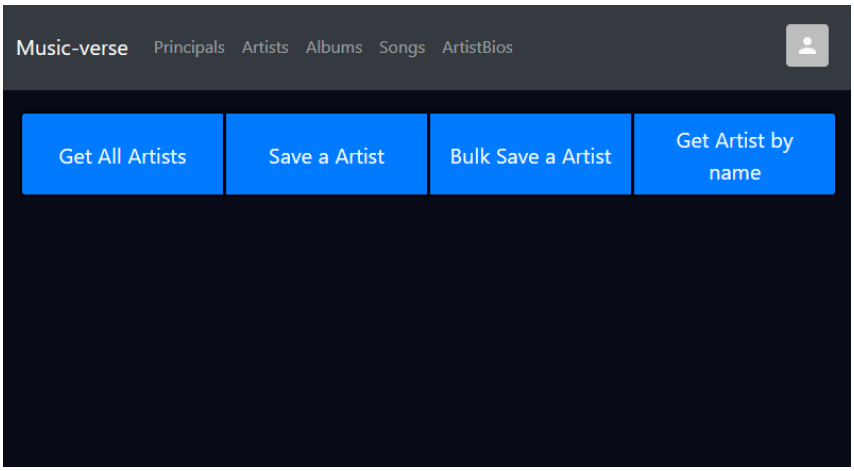

Figure 5-4: Example of a component screen

NavBar.jsx

The navigation bar component is an implementation of “reactstrap” navbar component. It has a default routing link and links to all other models and a button on the far right for logout.

ProfileNav.jsx

This component is the button on the far right of the navbar component and is a dropdown with two options: a link to the profile page of the user and a link to logout from the session.

APICall.jsx

This component, in contrast to the other components discussed, is not a visual component. It does not get rendered anywhere on the UI. This is a component that holds 3 reusable methods (HTTP GET, POST and DELETE requests) used for interacting with the server, i.e. every call to the server is served through these methods. All the methods use the fetch API to make requests to the server and every request adds the JWT as a header to have it verified on the server end. “async” and “await” commands are used wherever there is a call to the server to handle them asynchronously.

FormsComponent.jsx

This component gives a form input interface to the user. This is a reusable component and is used heavily throughout the application for different purposes. It takes as props the form input fields to render into the UI and then iterates through these fields to add those fields as part of the form. It also takes other props attributes from the parent component. Figure 5-5 shows an example of the props passed down to this component from the “Artist.jsx” model component. It also supports a multipart file as part of the form input.

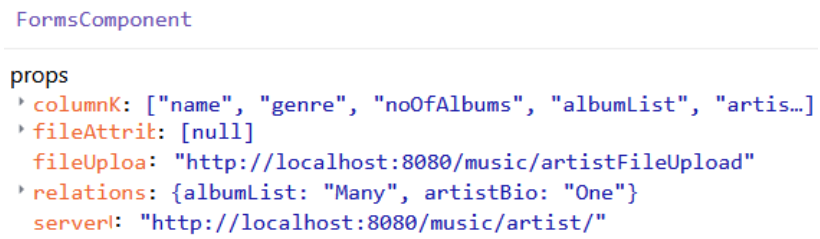

Figure 5-5: FormsComponent.jsx props example

TD.jsx

This reusable component is at the heart of an application like this where the bulk of the task is to interact with data, i.e. add, edit and delete it. This component takes care of rendering

the records into the UI for the user to interact with. It imports and implements the table from the globally available package “react-data-table-component”. This package provides a fresh and visually beautiful table which is customized for our purposes.

This table provides the functionality of pagination by clicking the corresponding dropdown button from the bottom right, along with buttons to navigate between the pages. Each record can be individually edited by clicking the corresponding “Edit” button, which renders a form component. The user can also select individual records or all records and then delete them by clicking on the delete button that slides in after a selection is made. The search box is customized for this application in such a way that it can take any input from the user and then iterate through all available records to find a match and then display only those records. All of this is done in the client end for it to be faster. This component can also render a picture for the models that have a file attribute. This is done by setting a special property to the column that carries the server location of the picture file. Since this is not rendered as a text in the UI, it is not sortable. Figure 5-6 shows a snapshot of props sent to this component for a render.

Figure 5-6: Data table props passed from Song model

CHAPTER 6. COMPARISON

Table 6-1 presents a comparison of the total number of features available in the previous version of the ADL against the refactored version of ADL.

Table 6-1: Comparison of features of both versions of ADL code base

Features	ADL previous version	ADL refactored version
Generate client app (parser)	Yes	Yes
Generate server app (parser)	Yes	Yes
Define entity relationships (server)	Yes	Yes
Authentication (client and server)	Yes	Yes
Add a record (client and server)	Yes	Yes
Edit a record (client and server)	Yes	Yes
Delete a record (client and server)	Yes	Yes
List all records (client and server)	Yes	Yes
Search a record (client)	Yes	Yes
Get record by name (client and server)	Yes	Yes
Sort records table (client)	Yes	Yes
Pagination of table UI (client)	Yes	Yes
Display picture (client and server)	Yes	Yes
Bulk upload records (client and server)	Yes	Yes
Download all records as csv file	No	Yes

As can be seen from Table 6-1, there is no loss of functionality during the refactoring process. We were able to improve upon some of the features, for example authentication, redesign the entire client end application template and also incorporate a new feature, which is to download all records from a table as a csv file. All the changes made to ADL code base has helped it be useful in a realistic sense, whereas it was in more of a proof of concept state in its previous implementation.

Table 4-1 in chapter 4 presents an analysis of the previous version of ADL code base with respect to the 5 standards of quality that we intended to incorporate. A similar analysis of the refactored version of the ADL is presented below in Table 6-2. Through the analysis we can prove that the refactored version of ADL has better code quality, which directly translates to a better user experience and this promotes future growth.

Table 6-2: Analysis of refactored version of ADL

Part of ADL	Cohesion	Coupling	Organization	Reusability	Extensibility
Parser	High	Loosely coupled	Organized	Reusable	Extensible
Server template	High	Loosely coupled	Organized	Reusable	Extensible
Client template	High	Loosely coupled	Organized	Reusable	Extensible

The next chapter provides an insight on our experiences and observations and discusses the scope of future work in this field.

CHAPTER 7. CONCLUSION AND FUTURE WORK

Our work serves to be a case study or a reference for the experiences in refactoring a poorly written code base. Upon close analysis of the existing ADL code it was evident that the quality of current code creates technical debt on the contributors to ADL. Through our refactoring efforts, we were able to generate a newer version of ADL that is modular and extensible. We were also able to introduce certain new functionalities to ADL to prove that future development will be significantly less resource hogging and will provide a better user experience for the end user and also encourage growth. In this chapter we provide a list of experiences and observations made during the entire refactoring process.

Observations

Inheriting a poor quality code requires more resources for change

The quality of code inherited was bad, hence it required significantly greater time to analyze and refactor. This is why it is important to incorporate best practices of software development as a developer.

Time constraints leads to poor design choices

Software development is a very big process involving many steps and every step should be given the same level of priority because of its complexity. Time constraints in development inherently leads to shortcuts and poor design choices and the quality of code takes a hit.

Verification of system functionalities after refactoring is important

Refactoring process should be promoting future growth, but this growth should not be at the cost of current functionalities. Hence, it is important to always verify that there is no loss of functionalities through this process.

Refactoring is a periodic process

Any software system requires periodic maintenance. The same way, it is important to periodically review and refactor the current code base. This helps to always keep the code up to date and comply with good software development practices, so that it does not become legacy code, which is significantly harder to refactor because of a multitude of factors.

Standardization is important for a shared workspace

JavaScript is both a very powerful language and difficult to maintain. In a shared workspace, it is important to introduce some form of standardization (like in Java) because different developers have different preferences and bias towards development and it is important to maintain the quality of code in spite of these differences. It also helps with readability as changing the modules will not require a reader to change the context of understanding because of similarities in implementation strategies.

Hard coded values should be avoided at any cost

Using hard coded values for anything in the code base leads to extremely tight coupling between the modules, which in turn makes it very tough to incorporate any change. Eliminating the hard coded values is a top priority for any refactoring process.

This work presented a report on the experiences of refactoring a poor quality code base from a developer's perspective, along with sharing the different types of refactoring implemented to reduce the technical debt. However, the refactoring reflects the preferences of a single developer. This is a very small sample space and any extension to this work will require a contribution from more developers. It will also be very interesting to consider the different backgrounds and cultures of the developers to understand the influences of such factors in the refactoring expectations of the developers.

REFERENCES

[1] Ghosh, Tanmay Kumar. “APP DESCRIPTION LANGUAGE,”

[2] Pandian, Preethi. “Application Description Language v1.2”

[3] Abelson, Harold, and Gerald Jay Sussman. Structure and Interpretation of Computer Programsnull. The MIT Press, 1996.

[4] Roberts, Don, John Brant, and Ralph Johnson. “A Refactoring Tool for Smalltalk.” Theory and Practice of Object Systems 3, no. 4 (1997): 253–63.

[5] G. Szóke, C. Nagy, R. Ferenc and T. Gyimóthy, "Designing and Developing Automated Refactoring Transformations: An Experience Report," 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, 2016, pp. 693-697, doi: 10.1109/SANER.2016.17+

[6] Schuts M., Hooman J., Vaandrager F. (2016) Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report. In: Ábrahám E., Huisman M. (eds) Integrated Formal Methods. IFM 2016. Lecture Notes in Computer Science, vol 9681. Springer, Cham. https://doi.org/10.1007/978-3-319-33693-0_20

[7] Roberts, D.B. and Johnson, R., 1999. Practical analysis for refactoring. University of Illinois at Urbana-Champaign.

[8] Fowler, M., 1997, June. Refactoring: Improving the design of existing code. In 11th European Conference. Jyväskylä, Finland.

[9] Tichelaar, S., Ducasse, S., Demeyer, S. and Nierstrasz, O., 2000, November. A meta-model for language-independent refactoring. In Proceedings International Symposium on Principles of Software Evolution (pp. 154-164). IEEE.

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," Proceedings Seventh Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, 2000, pp. 98-107, doi: 10.1109/WCRE.2000.891457.

[11] Ivan Moore. 1996. Automatic inheritance hierarchy restructuring and method refactoring. SIGPLAN Not. 31, 10 (Oct. 1996), 235–250. DOI:https://doi.org/10.1145/236338.236361

[12] K. Maruyama and K. Shima, "Automatic method refactoring using weighted dependence graphs," Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), Los Angeles, CA, USA, 1999, pp. 236-245, doi: 10.1145/302405.302627.

[13] Sang-Uk Jeon, Joon-Sang Lee and Doo-Hwan Bae, "An automated refactoring approach to

design pattern-based program transformations in Java programs," Ninth Asia-Pacific Software Engineering Conference, 2002., Gold Coast, Queensland, Australia, 2002, pp. 337-345, doi: 10.1109/APSEC.2002.1183003.

[14] Rocco Oliveto, Malcom Gethers, Gabriele Bavota, Denys Poshyvanyk, and Andrea De Lucia. 2011. Identifying method friendships to remove the feature envy bad smell (NIER track). In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). Association for Computing Machinery, New York, NY, USA, 820–823. DOI:https://doi.org/10.1145/1985793.1985913

[15] M. O. Cinneide, "Automated refactoring to introduce design patterns," Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium, Limerick, Ireland, 2000, pp. 722-724, doi: 10.1145/337180.337612.

[16] Beck, K., Fowler, M. and Beck, G., 1999. Bad smells in code. Refactoring: Improving the design of existing code, 1, pp.75-88.

[17] Fowler, M., 2000, July. Refactoring. In TOOLS (34) (p. 437).

[18] Fernandes, E. “Stuck in The Middle: Removing Obstacles to New Program Features through Batch Refactoring.” In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 206–9, 2019. https://doi.org/10.1109/ICSE-Companion.2019.00083.

[19] Schuts, Mathijs, Jozef Hooman, and Frits Vaandrager. “Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report.” In Integrated Formal Methods, edited by Erika Ábrahám and Marieke Huisman, 311–25. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-33693-0_20.

[20] Dams, D., A. Mooij, P. Kramer, A. Rădulescu, and J. Vaňhara. “Model-Based Software Restructuring: Lessons from Cleaning up COM Interfaces in Industrial Legacy Code.” In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 552–56, 2018. https://doi.org/10.1109/SANER.2018.8330258.

APPENDIX. ENVIRONMENT SETUP

The three parts of ADL code require different environment setups. We configured the IntelliJ IDE platform to refactor the ADL parser code. At the time of development, we used the IntelliJ IDEA (Community Edition) version 2020.2.3 along with JDK version 11.0.1 to build and JRE version 18.9 to run the application. Gradle build tool version was updated in the project from 4.4 to 5.6.3 to work with JDK 11.

The server application code that will be later used to create templates out of is developed in Spring Tools Suite version 4.5.1 for Eclipse. The server application uses Maven build tool and the version used is 2.3.4.

For developing the refactored client end code, we have used Visual Studio Code version 1.51.1. Node.js version 12.13.0 was used as our runtime environment, which uses Chrome's V8 engine. The package manager used is Node Package Manager (NPM) version 6.13.0, which comes bundled with Node.js. Below is a list of all the node packages that we used for the development of the client end code.

Node Packages

@material-ui/core

Material UI package contains many useful components that provide very helpful functionalities (icons, custom inputs, menu bars etc.) to develop a React JS application. The version used is 4.9.4.

Bootstrap

This is a very popular CSS and JavaScript based package that helps develop a mobile friendly web application. The version used is 4.4.1.

Reactstrap

This is a component based implementation of bootstrap functionalities that can be used in React JS development. The version used is 8.2.0.

React-data-table-component

ADL generates an application that is mainly used for manipulating data. This requires it to have a very accessible front end UI for the user to interact with. This package contains components that help render a functional table, which can be further extended with custom functionalities. The version used is 6.3.1.