A User Interface for adding Machine Learning tools into GitHub

Rumyana Rumenova Rafailova

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2015

Abstract

The Machine Learning group at Edinburgh has developed tools for automatic code summarisation, which takes the source code of a software project and decides on boilerplate blocks to be folded away. The aim of this project is to develop a web application which uses this tool to display summarised source code, and allows programmers to skim through GitHub projects at different levels of detail.

This paper describes the design, implementation and evaluation of the web application developed to this aim.

Acknowledgements

I would like to thank Charles Sutton for all his help and understanding throughout this project, Jari Fowkes for his prompt assistance with MAST issues, and Jane Hillston for being an excellent personal tutor, even during her sabbatical.

Table of Contents

Chapter 1

Introduction

The overall goal of this project is to provide a web application interface to the existing MAST (Machine Learning for Analysis of Software Text) tool, through which programmers can quickly familiarise themselves with a project on GitHub by browsing the files in it and looking through an automatically summarised version of its code.

Building on a set of requirements outlined in section 2.2, a web application was designed, implemented, and evaluated in iterative steps. Suitable technologies were chosen and explored (section 3.1), the MAST tool's function was understood on a high level (section 3.5.1) and means of communication with it were chosen upon. A suitable system structure was designed (section 1.3.2), and modified if necessary during development, to produce a modular, usable and maintainable system.

Implementation was done with the aim to periodically deliver a functional application in accordance with requirements in their order of importance, by substituting some functionality with stub methods that mimic its behaviour if necessary. The development process went from building an editor which displays code (section 3.2.1), to one which folds code blocks as instructed by the MAST tool (section 3.3.2). A project browser was then added (section 3.3.3), as well as features such as compression rate selection (sections 3.1.1 and 3.2.2) and improved usability via URL rewriting (section 3.4.3). The basis for user feedback (3.2.2) and improved project browser usability (section 3.3.3) was developed.

Evaluation against functional requirements is described in section 2.2.3. For non-functional requirements, GUI and regression tests were performed. They are presented in sections 4.2 and 4.1 respectively.

Much of the work on this project, being the development of a web application, was about exploring different software packages, choosing the right ones and making them work together. As these provide many different ways of achieving the same effect, specific functionalities were sometimes re-implemented using different libraries or language features. This is discussed throughout chapter 3 of this report, with reference to specific elements of the implementation.

1.1 Motivation

When deciding on whether to use a piece of open-source software, for example, a developer may want to get a sense of what algorithms and data structures are used in the project and what coding standards it satisfies, or to get a high-level understanding of its core functionality. The amount of time needed to achieve this could be significantly decreased if code is summarised with non-essential blocks folded away. Furthermore, automated code summarisation solves the chicken-and-egg problem of having to understand the code in order to ease the task of understanding it, because without automation, a developer must first decide which regions to fold. [14]

GitHub's code viewer, however, does not allow code folding. Although additional tools such as browser extensions or web applications may exist that do, they are difficult to find, and it is reasonable to assume that they would only provide manual folding as autofolding is not currently a feature even in full-fledged modern IDEs, except for trivial decisions such as folding imports in Java. [14]

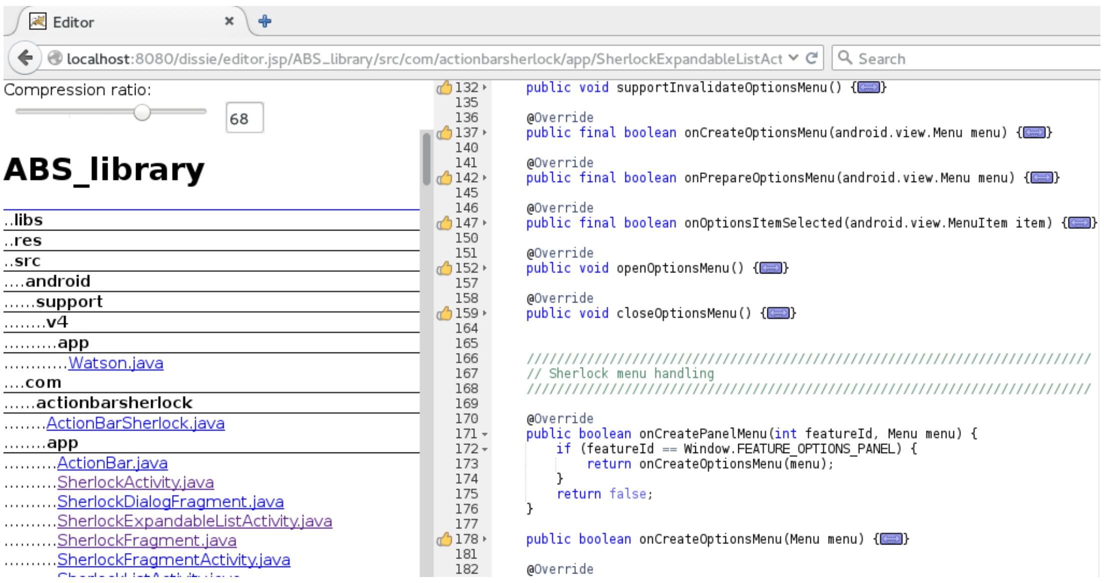
1.2 Usage

The web application built to this aim takes a URL such as `http://thisproject.com/crakeron/ABS_Library/blob/master/src/com/actionbarsherlock/app/ActionBar.java` for the file `ActionBar.java` in the GitHub repository `ABS_Library` by `crakeron`, and displays a summarised version of it, together with a project browser where another file of interest can be selected.

This format was chosen because of its simplicity for the end-user: the same effect is achieved by substituting github.com with thisproject.com on a specific file's URL like `https://github.com/crakeron/ABS_Library/blob/master/src/com/actionbarsherlock/app/ActionBar.java` Further information on this can be found in section 3.4.3.

Figure 1.1 shows a general view of the web application. A source file's code is displayed in a style similar to that of an editor, with syntax highlighting and line numbers.

By default, code is summarised with its non-essential blocks folded away, at a default ratio of 50 (out of 100). The user can manually unfold regions to get more detail, or change the level of granularity by adjusting the compression ratio used by the MAST (Machine learning for Analysis of Software Text) tool, so that the whole file is automatically summarised to a different level of detail.



localhost:8080/dissie/editor.jsp/ABS_library/src/com/actionbarsherlock/app/SherlockExpandableListAct

Compression ratio: 68

ABS_library

..libs
..res
..src
....android
......support
........v4
..........app
............Watson.java
....com
......actionbarsherlock
........ActionBarSherlock.java
........app
..........ActionBar.java
..........SherlockActivity.java
..........SherlockDialogFragment.java
..........SherlockExpandableListActivity.java
..........SherlockFragment.java
..........SherlockFragmentActivity.java

```
public void supportInvalidateOptionsMenu() {↔}

@Override
public final boolean onCreateOptionsMenu(android.view.Menu menu) {↔}

@Override
public final boolean onPrepareOptionsMenu(android.view.Menu menu) {↔}

@Override
public final boolean onOptionsItemSelected(android.view.MenuItem item) {↔}

@Override
public void openOptionsMenu() {↔}

@Override
public void closeOptionsMenu() {↔}


///////////////////////////////////////////////////////////////////////////
// Sherlock menu handling
///////////////////////////////////////////////////////////////////////////

@Override
public boolean onCreatePanelMenu(int featureId, Menu menu) {
    if (featureId == Window.FEATURE_OPTIONS_PANEL) {
        return onCreateOptionsMenu(menu);
    }
    return false;
}

public boolean onCreateOptionsMenu(Menu menu) {↔}

@Override
```

Figure 1.1: The web application.

1.3 Realisation

1.3.1 Overview

To display code, the web application loads the necessary Java file from the server's file system, which has a number of popular GitHub repositories cloned on it (this is further discussed in section 3.5.2). Code presentation is a feature of Ace editor, which was used for this project (more information in section 3.2.1).

For the project browser (on the left side of figure 1.1), the application generates an HTML "tree" representing the current project's files of interest (in this case, Java files) by using Java's file API (discussed in section 3.3.3).

For summarisation, it calls a method from the MAST tool which provides it with the line numbers of regions to fold. The actual folding is implemented in JavaScript by making use of Ace editor's manual folding function.

1.3.2 High-level Design and Implementation

The system's core functionality is built using JSP (introduced in sections 3.1 and 3.3.1) on top of Tomcat (introduced in sections 3.1 and 3.4.1), which generates HTML and JavaScript for the front end, and communicates with the MAST tool API at the back end. This is illustrated in Figure 1.2.

User interaction with the system typically begins at `editor.jsp`, which includes the pages responsible for editor and code browser functionality - respectively, `fold.jsp` and `browse.jsp`. A request to editor.jsp instructs Tomcat to compile JSP pages into

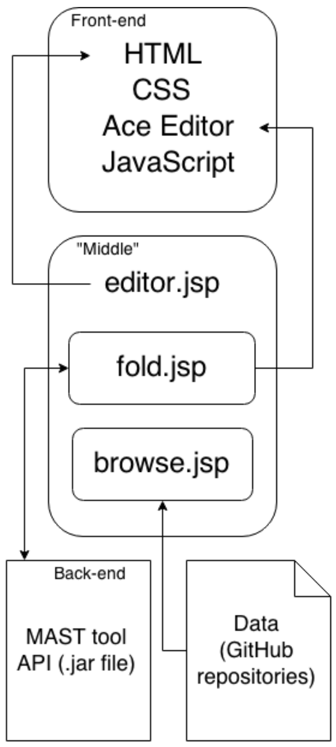


Figure 1.2: High-level system design.

servlets and run them on the server, which generates the HTML and JavaScript and sends it to the browser.

The web application starts with displaying a hard-coded default GitHub project and source file, with default compression ratio of 50 (out of 100).

The servlet compiled using `browse.jsp` is responsible for the file browser (on the left in figure 1.1). It generates the HTML for a file tree of GitHub repositories cloned to the server: in this prototype, all of them are displayed. Section 3.3.3 discusses browse.jsp in more detail.

The other servlet, compiled using `fold.jsp`, is responsible for the (read-only) code editor (on the right in figure 1.1). It starts by calling a method out of the MAST tool back end to receive the lines to be folded at the current compression ratio. It then generates some JavaScript to populate a div with the source code of the given file by preparing a GET request for the actual file, via an `XMLHttpRequest()`. The servlet uses the information sent by the back end to generate JavaScript code for the code block folding, by calling methods provided by Ace editor. The servlet then declares some JavaScript functions which are responsible for the user feedback functionality (described in section 3.2.2). Section 3.3.3 discusses fold.jsp in more detail.

All of this is generated as HTML/JavaScript by the servlet, and sent to the browser for execution. As GET requests, editor setup via Ace libraries, and other JavaScript functionality are executed asynchronously, programmatic folding of code blocks is only called after a certain (minimal) time-out.

Chapter 2

Design

2.1 Work Process

Work on this project began by discussing rough requirements and project aims, getting a general idea of what the research which produced the MAST tool is about, and clarifying the motivation to produce the web application interface. A general plan of how to organise the project was agreed upon.

The next step was to familiarise myself with the MAST tool, which would serve as a back end for the web application. Although it is now used as a black box, it was not initially clear how to achieve this (discussed in section 3.5.1). Being a large code base, and new for me, meant it took some time to get it running, and to understand which parts of it the web interface would need to communicate with.

An informal design document was produced, including Planning, Implementation, and Code process, plus technologies to look into. We discussed use cases, interface behaviour, and steps to aim for depending on how quickly things get done: from ”primary” to ”stretch” requirements.

The initial idea was to develop a JavaScript code browser similar to a simplified version of the one on GitHub. In researching this, Ace - a flexible open-source JavaScript code editor - came up as a viable option to use (it is discussed in more detail in section 3.2.1). Thus, the project’s focus could be shifted forward in our set goals. This did not prove to be a problem as our plan already included additional requirements to extend core functionality as time allows.

From then on, work on the project has consisted of iterative steps to implement functional elements of the web interface (further discussed in section 2.2.1).

2.2 Requirements

An agile methodology was adopted in working on this project, so requirements were specified by starting with the minimal functional elements and building up extra features on top. The following list is ordered accordingly, in order of the importance of these requirements for overall system functionality. It represents the later stages of system design.

2.2.1 Functional

1. Browse an individual file.

   A read-only code editor should display the file of interest, in a format similar to that of a highly simplified version of the GitHub code browser. It should display the code with correct indentation, and allow the user to request more detail by manual unfolding. The primary requirement for this is that manual folding could be done by code region. The stretch goal is to allow the user to selectively unfold identifiers of interest.

   The code displayed should be summarised by having code blocks which are non-essential to understanding its functionality folded away. Folding should first be done at a default level of detail (compression rate).

   The user should then be able to change the level of detail and receive a new view of the code with different folding. As a primary requirement, this should be done statically (by reloading the page), and as a stretch goal - dynamically.

   Once this is done, a useful feature would be to allow users to provide feedback of the summary they have interacted with. Primarily, this should be a rating of how useful they find the full file's folded version. As a stretch goal, it should be possible to provide feedback on separate code regions that have been folded by the system.

2. Browse the files in a GitHub project.

   Once there is working functionality on individual file browsing, users should be able to navigate to other files within the same GitHub projects. The primary requirement for this is to provide a view which resembles file structure. As a stretch goal, the system should provide a visualisation of the whole project in terms of ”interesting” files.

3. Select a GitHub project.

   Users should the able to browse the files of other GitHub projects in a similar manner as described above.

2.2.2 Non-functional

The main non-functional requirement for this project is usability. In terms of compatibility with major browsers, the technologies used were chosen accordingly (section 3.1), and evaluation was done in multiple browsers (section 4.2). Correct behaviour in response to user actions was evaluated in section 4.1.

Response time was also briefly examined in section 4.1, as a feature of usability.

2.2.3 Requirements evaluation

In terms of the code editor, implemented functionality has met the requirements for being read-only, well-indented and allow manual folding on the user's part (these were achieved by using Ace, as discussed in section 3.2.1). Further features are available through it, such as syntax highlighting and showing the occurrences of an identifier by double-clicking on it. Unfolding selected identifiers of interest was not implemented. A suggestion on how this can be done in the future is outlined in section 3.7.

Automated folding functionality is fully implemented. It is described in sections 3.2.1 and 3.3.4.

User changes to compression ratio have been implemented statically, with added JavaScript functionality to mimic dynamic behaviour. The reason for this is discussed in section 3.2.2.

As user feedback on the entire folded document was determined to be less useful than feedback on code regions, it was not implemented. Only the front-end functionality for block-level feedback was implemented. This is detailed in section 3.2.2. Section 3.7 contains a note on a possible back end.

Project browser functionality was implemented as described in section 3.3.3. Currently all repositories available on the server are listed in the file browser. In a future implementation, if there is a separate page which lists available projects, or the application will be used primarily via the URL rewriting feature (previously discussed in section 1.2), it may be more useful to only display the current project's files. This can be done by simply changing the string in `browse.jsp` which represents the root directory, where the file tree starts looking for children to generate into HTML.

Full-project visualisation of interesting files was not implemented.

Selecting another GitHub project is currently possible by just choosing a file from it out of the file browser.

Chapter 3

Implementation

3.1 Technologies used

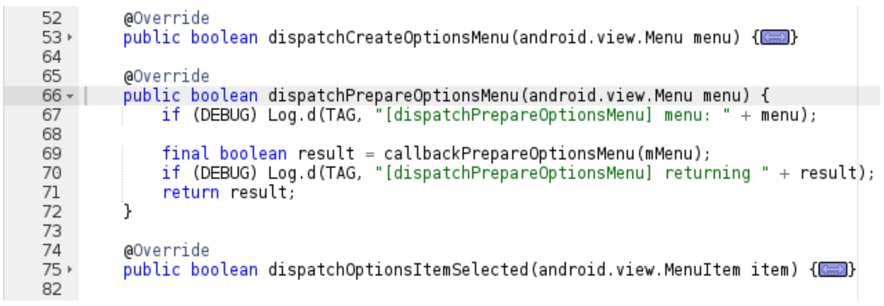
- `HTML/CSS` - for display
- `JavaScript` - as part of the Ace editor, and for other front-end functionality (section 3.2.2)
- `JSP` (JavaServer Pages: a technology to create dynamically generated web pages, part of Java EE) - to communicate with the MAST tool, to provide the project browser (section 3.3)
- `Tomcat` (a web server and servlet container implementing JSP) - to run the web application, debug it, and rewrite URLs for added usability (section 3.4)
- `Java` - in JSP (section 3.3), and to understand, and make small modifications to, the MAST tool back end (section 3.5.1)

3.1.1 HTML

Most of the HTML used is HTML4 compliant, with the exception of HTML5's input slider element, used for choosing a compression ratio. As this element has been supported in all major browsers for a while [3], and the target audience of the web application are programmers, it was chosen as a good alternative to much heavier ways of achieving the same functionality (such as CSS+JS in jQuery or other external libraries). A separate JS function was written for full compatibility with some versions of Firefox. For all older browsers, the option of manual input is still available.

3.1.2 JSP/Tomcat

As the MAST tool back end is written in Java, dynamic communication with it is most easily achieved by using a Java web technology. The suitable framework for this



```
@Override
public boolean dispatchCreateOptionsMenu(android.view.Menu menu) {↔}

@Override
public boolean dispatchPrepareOptionsMenu(android.view.Menu menu) {
    if (DEBUG) Log.d(TAG, "[dispatchPrepareOptionsMenu] menu: " + menu);

    final boolean result = callbackPrepareOptionsMenu(mMenu);
    if (DEBUG) Log.d(TAG, "[dispatchPrepareOptionsMenu] returning " + result);
    return result;
}

@Override
public boolean dispatchOptionsItemSelected(android.view.MenuItem item) {↔}
```

Figure 3.1: Code displayed with indentation and syntax highlighting.

project thus needed to be able to run Java and provide a communication mechanism for the results returned by the MAST API. The finished project's requirements to the environment underlying it is to be able to run a .jar file, and call a method from it to receive integer output (implemented in the JSP). As this was not clear from the start, a framework with maximal Java flexibility was chosen.

There is an overwhelming array of options for Java web programming in terms of standards, best practices and environments - according to Wikipedia, there are 38 different frameworks for Java [17]. My initial research showed that JSP seems to be the most popular technology among extensive Java web development articles and tutorials. JSP was chosen over direct servlet programming as it provides an abstraction over it, and should be better suited to ”views”, or close-to-front-end functionality, which is the aim of this project.

Due to Tomcat's leading popularity among Java application servers [2] and its focus on JSP [15], while other technologies such as JSF are not implemented by it, I decided upon a JSP/Tomcat environment.

The choice of JSP/Tomcat is evaluated in section 3.6.1.

3.2 Front end

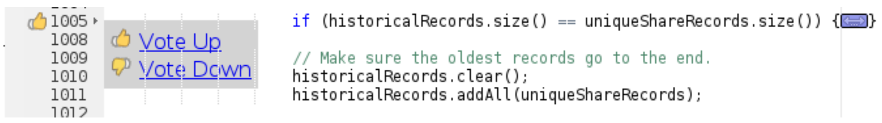
3.2.1 Ace editor

Ace is an easily customisable embeddable open-source code editor written in JavaScript. The initial project plan was to develop a highly simplified version of the GitHub code browser, and GitHub actually uses Ace [8]. The decision to use it instead of developing a new one allowed initial ”stretch” goals for the project to be promoted to ”primary”, as presented in section 2.2.1. As Ace is a popular and active system, it works correctly in major browsers and is less prone to problems than a newly built editor.

In the web interface, Ace provides all code display functionality, including syntax highlighting, line numbers, and manual folding. (figure 3.1)

All it takes to use it is the following code:

```
<div id="editor"></div>
```



```
if (historicalRecords.size() == uniqueShareRecords.size()) {}

// Make sure the oldest records go to the end.
historicalRecords.clear();
historicalRecords.addAll(uniqueShareRecords);
```

Figure 3.2: User feedback functionality.

```
<script src="ace.js" type="text/javascript" charset="utf-8"></script>
<script>
    var editor = ace.edit("editor");
    editor.setReadOnly(true);
    editor.getSession().setMode("ace/mode/java");
</script>
```

To fold code regions programatically, after initial attempts to modify Ace's source code, a solution using its standard functions was found [13]:

```
 window.setTimeout(function() {
     editor.getSession().foldAll(startLine, endLine, depth);
}, 100);
```

It is used in combination with JSP to fold the lines as determined by the MAST tool.

Some of Ace's more hidden functionality was used and adapted to provide the basis for a user feedback feature, which was previously noted in section 2.2.1 and is demonstrated in figure 3.2. The aim is for users to be able to vote on the application's decision of which code blocks are non-essential (and thus folded away at this compression rate).

As a code editor, Ace provides functionality used by systems that compile the code written in it, which allows some standard images to be placed next to the number of a line that gives an error or warning. The web application loads these at folded line numbers (further information in section 3.3.4). Ace's CSS for the icon was overwritten to display a "thumbs-up" icon. A hover-over tool-tip, usually used on identifiers, was added to this to display a "Click to vote" hint. Using JavaScript, these icons were made clickable and a div was added at the proper location to provide a context menu where users can vote the fold up or down.

3.2.2 JavaScript

In addition to being used to run and customise Ace, JavaScript was used to populate the editor with the necessary file's code via an `XMLHttpRequest`.

JavaScript functions were also written to provide an immediately visible and easy-to-use way for the user to view source code at a different level of detail - by changing the current compression ratio. This is done through an HTML5 slider element (discussed in section 3.1.1), and a text field which displays the current compression ratio and could be modified by the user. Interacting with these elements results in a page redirect via `window.location` to a JSP with the appropriate compression ratio parameter.

Figure 3.3: Functionality for changing the current compression ratio.

While this is a static solution to the problem of changing the compression ratio of a document, to the user it appears to be happening dynamically, as soon as they make a change using the slider or text field. As `fold.jsp` (discussed in section 3.3.4) needs to issue a new request to the back end (the .jar file providing the MAST tool API) and generate new JavaScript calls to fold the appropriate code blocks, reloading the whole page is a viable option. In this case, the Tomcat server spares its own resources by only re-compiling and re-running the servlets that are indeed changed (here, the one produced by fold.jsp).

Only Java files are recognised by the project browser and the editor, as they are the current focus of MAST, but this can easily be extended to other languages by adding a single JavaScript file for syntax highlighting (available as part of Ace), changing a variable against which Mime-type checks are performed in `browse.jsp` (discussed in section 3.3.3), and modifying Tomcat's rewrite configuration accordingly (discussed in section 3.4.3).

Different themes (out of many already available for Ace) for the editor could also be added with just the inclusion of an additional JavaScript file for the theme, and setting it up with `editor.setTheme("ace/theme/twilight");`

3.3 "Middle" (JSP)

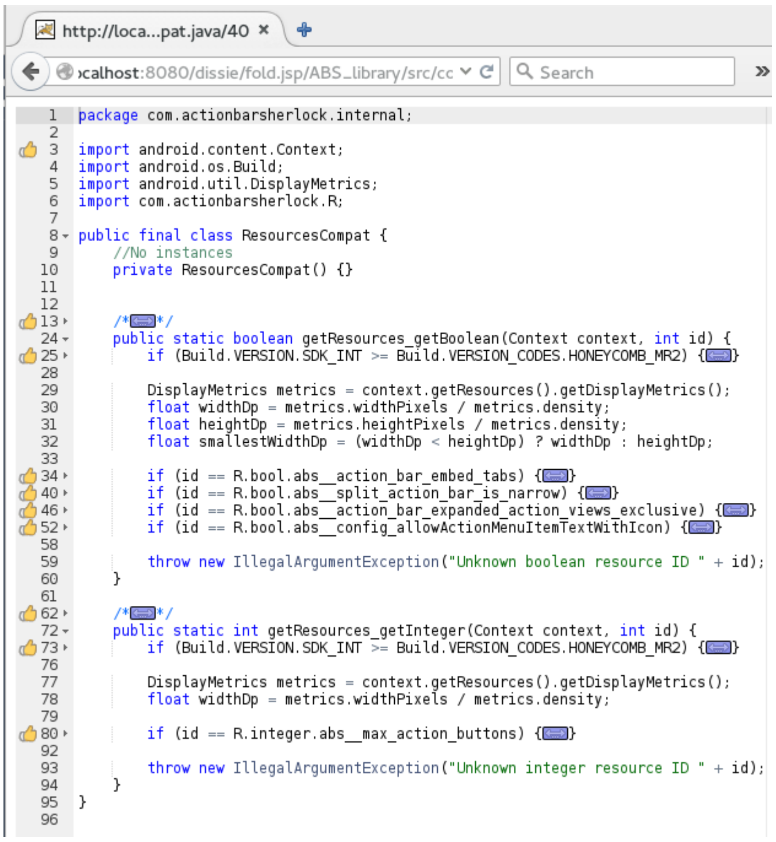
The web application's core functionality is implemented in JSP, which calls functions from the MAST tool API to receive information about which code areas should be folded, and generates the final HTML/JavaScript code for the web application.

3.3.1 Introduction to JSP

JSP is a Java view technology which provides a programming framework for web development. JSP files are compiled to generate Java code, and converted into Java servlets to be run on the server. This is done by the servlet container (in this case, Tomcat). The location of code in the generated servlets is determined by JSP directives, including

declarations, which become part of the class of the generated servlet;

scriptlets, which are inserted into the `service()` method of the generated servlet.


http://loca...pat.java/40
›calhost:8080/dissie/fold.jsp/ABS_library/src/cc
Search

```java
package com.actionbarsherlock.internal;

import android.content.Context;
import android.os.Build;
import android.util.DisplayMetrics;
import com.actionbarsherlock.R;

public final class ResourcesCompat {
    //No instances
    private ResourcesCompat() {}


    /*…*/
    public static boolean getResources_getBoolean(Context context, int id) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {…}

        DisplayMetrics metrics = context.getResources().getDisplayMetrics();
        float widthDp = metrics.widthPixels / metrics.density;
        float heightDp = metrics.heightPixels / metrics.density;
        float smallestWidthDp = (widthDp < heightDp) ? widthDp : heightDp;

        if (id == R.bool.abs__action_bar_embed_tabs) {…}
        if (id == R.bool.abs__split_action_bar_is_narrow) {…}
        if (id == R.bool.abs__action_bar_expanded_action_views_exclusive) {…}
        if (id == R.bool.abs__config_allowActionMenuItemTextWithIcon) {…}

        throw new IllegalArgumentException("Unknown boolean resource ID " + id);
    }

    /*…*/
    public static int getResources_getInteger(Context context, int id) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {…}

        DisplayMetrics metrics = context.getResources().getDisplayMetrics();
        float widthDp = metrics.widthPixels / metrics.density;

        if (id == R.integer.abs__max_action_buttons) {…}

        throw new IllegalArgumentException("Unknown integer resource ID " + id);
    }
}
```

Figure 3.4: fold.jsp on its own.

3.3.2 editor.jsp

The application's "start-page" `editor.jsp` is where the general HTML skeleton resides. It also includes the necessary style-sheets and scripts, and the functionality for the compression ratio slider (discussed in section 3.2.2).

Two main modules: `browse.jsp` and `fold.jsp`, are responsible for the project browser and code editor respectively. `editor.jsp` includes these dynamically via the `<jsp:include>` action, which compiles and executes each page as a separate servlet. This contributed to the modular design of the application, and allowed each page to be implemented with its complete functionality - using its own variables and methods, and making the appropriate checks for variables such as project/file name and compression ratio. browse.jsp and fold.jsp can thus function on their own and be reused as components. This also adds extra functionality, letting the user look at code only without the file browser. This is demonstrated in Figure 3.4.

3.3.3 browse.jsp

This file contains the functionality for generating the file browser found on the left-hand-side of the application's interface (which was shown in figure 1.1).

Generating the HTML for a file tree within JSP presented some challenges, requiring a solution which:

1. is custom, as there is no convenient out-of-the-box way to deal with this task.
2. overcomes the limitations due to JSP being compiled into servlets before it is run;
3. spares the server's resources.

In more detail,

1. Java provides a number of different APIs to deal with files, but the more recent and convenient solutions like Java 8's functional-style features over `Files.walk()` or Java 7's `Files.walkFileTree()` are not available in JSP/Tomcat.
2. JSP's standard object `out` provides the I/O interface between scriptlets and the generated HTML, and is necessary in order to populate the HTML with dynamically generated data - such as a representation of a file tree. As the `out` object is declared within the `service()` method where scriptlets are executed, it is not available in JSP declarations.
3. Thus, the entire HTML of the file tree needed to be generated before it was output using `out.print`. An obvious alternative would be to pass the `out` object as a parameter to all the necessary methods, but JSP creates a new `out` object upon each HTTP request. This meant that a resource-intensive operation - the recursive tree traversal - would be re-run unnecessarily. Just one user session can generate tenths or hundreds of requests on different source code files, which need to be re-compiled into the appropriate servlet. The final aim of this project is to deal with many large code-bases, and it needs to be responsive to multiple users.
   As divs have entirely replaced frames (which are not even supported by HTML5) on the web, the only feasible solution was to generate the file tree statically.

The solution was implemented as follows:

Using some of `java.io.File`'s basic methods, `browse.jsp` provides the methods to recursively traverse the server's file tree of GitHub repositories. It sorts files by name, takes empty and hidden directories into consideration, produces the correct indentation for the file tree structure, and adds links to the file's respective editor page.

In order to make the code maintainable, it is organised in the following methods, which pass the necessary parameters to each other (as they cannot use non-final class variables):

`listFileTree()` - the main recursive function which, starting from a given root directory, calls either

`dirHTML()`, if the current element is a directory, or

`fileHTML()` if the current element is a file.

A separate `levelIndentation()` function takes care of the visual presentation of indented files/folders, in this prototype shown as:

....com
......actionbarsherlock
........ActionBarSherlock.java

where dots can easily be changed to images or special characters.

During the later stages of the project, these methods were modified to produce correctly structured HTML where each directory's contents are within its own `<div>`, so that the user could fold/unfold directories of interest, making the webapp behave more like what is expected of a browser. The code for actually folding these is not yet added, but should be straightforward to implement because the structure for it is already there.

3.3.4 fold.jsp

This page contains the code editor functionality: receiving the appropriate folds from the MAST tool and generating the necessary JavaScript to be executed for this particular source code file.

After performing some checks and setting up some values, it makes a method call to the jar file containing the MAST tool API (further discussed in section 3.5.1), providing it with information about the file to be folded and the desired compression ratio. It receives an ArrayList of line numbers, which represent the beginning and ending of code blocks to be folded. It then generates calls to the appropriate JavaScript functions (previously discussed in section 3.2.2) for each of these code blocks. In order to keep the code clean, during this loop it also prepares an array of the line numbers where a feedback button should be displayed, and passes it to the JavaScript (previously discussed in section 3.2.1).

3.4 Tomcat

3.4.1 Introduction to Tomcat

Apache Tomcat is an open-source implementation of several Java EE specifications. In this project, it is used as a servlet container for JSP (meaning that it is responsible for the compilation of JSP pages and running of servlets), as well as a web server - it can also be used over the Apache web server in a production version of the application.

3.4.2 Configuration

The configuration options necessary to run the application were described as .xml files within its directory, according to Tomcat's structure. It is thus portable and can co-exist with other applications on a production server.

Throughout debugging, in getting the technologies involved to work together, more time was spent looking through Tomcat's configuration than was necessary for the final state of the project. Some problems specific to Tomcat emerged in running the MAST tool .jar file (further discussed in section 3.5.1), possibly due to version incompatibility. Although the latest version of Tomcat (8) was used throughout the project, it does not officially support Java 8, which the MAST tool uses. However, Tomcat 8 has been known to successfully run some of Java 8 [4], and in the case of this project, after some modifications to the MAST tool backend (section 3.5.1), has succeeded with running the .jar file.

3.4.3 URL Rewriting

In order to implement the usage described in section 1.2, the Tomcat server rewrites GitHub-like URLs to ones containing parameters. URLs like `http://thisproject.com/editor.jsp/ABS\_Library/.../ActionBar.java` are rewritten to `http://thisproject.com/editor.jsp?project=ABS_Library&file=.../ActionBar.java`. The user can additionally specify the desired compression ratio at the end as `http://.../ActionBar.java/70`. A similar effect is achieved when referencing fold.jsp instead of editor.jsp, which displays the source code file only. Trailing slashes are also handled.

This was implemented via the Tomcat `Rewrite valve`, which functions similarly to Apache HTTP Server's `mod_rewrite`: with conditions consisting of regular expressions, and rules for substitutions.

Special care had to be taken to make sure Tomcat does not rewrite URLs it should not, such as

- `.js` and `.css` files.

  As requests for these come from files whose URLs may have already been rewritten, using relative URLs within the application's code may result in incorrect requests. For example, a request for `js/editor.js` from `http://.../ABS_library/.../app/Watson.java` is interpreted as a request for `http://.../ABS_library/.../app/js/editor.js`.

  This was solved by explicitly rewriting the URLs for files in the `js` and `css` directories through Tomcat configuration.

- the original file whose source code needs to be included in the editor.

  The application needed to disambiguate between URL rewriting cases, and calling `http://.../Watson.java` when populating the editor via a an `XMLHttpRequest()` in `fold.jsp`.

  This was solved by using an absolute URL in `fold.jsp` (which is generated by the JSP and is thus still portable).

3.5 Back end

3.5.1 Communication with the MAST tool

After a discussion on what output this project would need, Jari Fowkes, who is the original author of the relevant part of the MAST tool (written in Java), wrote an API that made it possible to call code summarisation as a black box through a command line interface. Afterwards, decisions could be made on the specifics of input and output necessary to build the web interface. Initially, the API was used through command line to produce sample outputs which were hard-coded into the application prototype.

Due to JSP/Tomcat being new technologies for me, it took some time to arrive to the conclusion that using a pre-packaged jar file and calling a function from it in JSP was the optimal option for communicating with the back end. Again with the help of Charles and Jari the project arrived at a state where the API's Java code could be modified and then easily re-built using maven. Modification on my part was necessary for two reasons:

- To suit the web application better - where arrival to this conclusion was only clear after the API was already written - for example, outputting only the start and end line of folded regions instead of a list of all the lines that should be hidden in the summary.

- To correct errors that only came up when using it through Tomcat - like the JVM being unable to run dependencies that are not really needed by this particular API, or logging to a file or to standard output which is done somewhere else in the MAST tool.

3.5.2 Data (GitHub repositories)

The back-end summarisation tool works by first building a model of all the included repositories and storing it in a file, which can remain unchanged as long as there are no changes to the repositories.

The actual code bases are in git repositories cloned on the server. A number of popular GitHub repositories are stored locally and used for the web application. This decision makes the project feasible resource-wise, including traffic and processing for model-building.

Coping with changes in repositories was not implemented in this project. This is discussed in section 3.7.

3.6 Challenges

As this project was about developing a web application, much of the work on it consisted of finding the right technology, learning how to use it and integrating it with the others.

At times - especially in the beginning when my experience was very limited - debugging proved to be a challenge because it often was not straightforward to understand which part of the system an error was coming from.

As the MAST tool is an active project, relying on it as a back end required some flexibility in system configuration. This was alleviated by packaging the necessary functionality into a jar file, which after setting up to work suitably with the web interface, did not need further changes (more information in section 3.5.1).

3.6.1 Choosing JSP/Tomcat

Tomcat is to a large extent a production system, and its extensive configurability can be at odds with learning. Setting it up to keep helpful log files, for example, took a long time because of the differences in Tomcat versions and OSs, and the necessary compatibility between debug options on different levels of configuration.

JSP was released in 1999 and since then has evolved to incorporate different programming styles, on the usage of which there does not seem to be a general consensus in the community. This makes learning it an exercise in agility, and in my experience was contrary to productivity.

Many beginner's tutorials on Java web programming focus on JSP: because it has been around the longest, or because it provides a simple way to get started by mixing Java code with HTML. Although JSP has now been deprecated by Oracle [5], top results on their website search about web development point to articles about JSP or servlets [12], [11]. Closer inspection throughout the project showed that a lot of the documentation available on JSP is outdated, and it is often difficult to find recent resources on more specific topics.

Further to difficulties in finding recent documentation, some versioning issues (detailed in sections 3.4.2) could have been avoided by using a different technology. JSP's code base does not seem to be active [10]. Using Java 8 for compatibility with the MAST tool together with JSP, whose last official mention in the Java EE tutorial was at version 5 [7], was hardly an ideal configuration.

As the application's core functionality was built in JSP, though, it did not seem wise to start from the very beginning with a completely different technology. Only later, with some knowledge of the Java web programming technology landscape, I was able to find resources detailing newer and more recently-documented technologies such as JSF over Apache MyFaces. Therefore, with the experience gathered from working on this project, I would now choose different technologies for it.

JSF is a web application framework based on the Model View Controller (MVC) paradigm. Separation of presentation, control, and data is an integral part of it [6], as it is organised around (and provides) reusable UI components [16] which are represented, managed, and connected via the JSF API [4]. JSP/Servlet programming, on the other hand, uses pages which contain a whole range of functionalities. Their structure and the connections between them need to be designed and debugged by the programmer. [9]. Although this should in theory mean more flexibility, the implementation of browse.jsp, which strived to be maximally modular, ended up more complicated than it needs to be because of some JSP limitations, as described in section 3.3.3.

JSF was specifically designed with the goal of simplifying the development of web-based user interfaces. Since Java EE 6, it provides a different declaration language (Facelets) [5]. The increased framework support for extensible components would allow for more meaningful debug messages: it is hard to say whether this is actually the case without using it to develop a full-fledged web application.

3.7 Suggestions for future work

In order to make this web application usable in practice, it should first be hosted to a web server. This was not done in the duration of this project because a recent version of Tomcat was not available in the free web hosting services researched. One option which was more closely examined was OpenShift.com, as it allows more configuration freedom. However, the attempts to run the application on it were not successful, as the latest officially supported version of Tomcat on it was 7, and setting up Tomcat 8 to work with Java 8 under this environment was unsuccessful (as previously discussed in section 3.4.2, Tomcat 8 has no official support for Java 8).

To cope with changes to repositories, a separate process on the server would need to poll git repositories to see if there are any changes, and retrain the MAST tool's model accordingly.

In terms of the system features discussed in this report, the feedback option described in section 3.2.2 has its front-end functionality in place, but the back end for it was not implemented. For feedback reports to be useful, ideally they would need to be stored in a database. For extended file browser functionality, as discussed in section 3.3.3, the HTML file tree is generated as a series of nested `<div>` elements, and adding hide/show behaviour to them via JavaScript would improve user experience.

As Ace has a notion of identifiers (as evidenced when double-clicking on one), implementing selective unfolding of identifiers of interest should be possible to achieve with customised JavaScript over Ace's functionality, similarly to what was described in section 3.2.2 for feedback functionality. A useful resource on that is the Ace Editor Google Group [1], which provides information on undocumented functionality and has a helpful community.

The code for this project was written with maintainability in mind (as noted throughout chapter 3), and changes like these are not expected to pose a significant difficulty.

Chapter 4

Evaluation

4.1 Regression testing

As this project is a simple web application, and the set of control flow paths in it follows a fixed structure, full-fledged testing frameworks were decided against. Instead, simple automated tests with minimal overhead were performed.

In order to test the correct behaviour of the system, the generated HTML was chosen as the baseline for evaluation. If any JSP component of the application fails, either a server error or a mis-generated HTML/JavaScript would occur. Test cases were thus designed as URLs that should point to available projects/files at differing compression ratios.

Test case URLs were generated with their expected correct HTML. These were then compared with the actual server output using wget.

This process consisted of the following steps:

1. Generate URLs. A Python script was written to traverse the file tree of cloned repositories and generate URLs for each .java file in them, in the form of `http://localhost:8080/dissie/editor.jsp/ABS_library/src/android/support/v4/app/Watson.java`
   For each URL, the following mutations were added to the URL list:

   the URL with a trailing slash

   the URL with trailing /x where x is a compression ratio, with a mutation for all integer numbers between 0 and 100.

2. Get the line numbers of blocks to be folded at given compression ratio. A bash script was written to run the MAST tool API for the given file and compression ratio. As the API was written to allow a command-line interface, this could be done by calling a method out of the same .jar file that the application uses. Folds were saved into the file folds.txt and then fed to a Python script to generate correct HTML.

3. Generate expected HTML for each URL. Much of the HTML is the same for all files. Differences arise due to the file name in the function `populateEditor()` and the parameters to the folding function. These which were generated by this script to arrive at a template.html file.

4. Compare expected to actual HTML. A bash script was then written to call the URLs from the localhost server via wget, record their response into response.html, and then compare it with template.html using diff.

Tests were performed on three popular GitHub repositories written in Java: ABS_Library, ActionBarSherlock, and androidassynchttp. Due to the large time required to run tests with full coverage of the range of compression ratio, files in these repositories were tested at a randomly chosen compression ratio in the range 0-100. 5 randomly chosen files from each repository were tested against all compression ratios in the range 0-100. Further tests will be run after the writing of this document.

Results showed incorrect behaviour when special characters (such as -) were used in some directory names. This was due to an inexhaustive pattern in Tomcat's URL rewriting configuration, which was fixed accordingly.

Examining Tomcat logs for these experiments showed no significant stall in response time. In terms of memory usage, no leaks were found and memory limit was not hit. For a realistic evaluation of response time, the application needs to be deployed on a server with production settings which allows simultaneous requests from multiple users.

4.2 GUI testing

As this project adopted an agile methodology, the web application's behaviour was tested manually throughout the system implementation as each new component or feature was developed.

As it is difficult to design automated tests that simulate user actions without a heavy testing framework, the application was tested manually to ensure that possible sequences of user actions are handled correctly. These included combinations of:

- navigating to a new file within the current project or another project,
- changing the URL to a different file or compression ratio,
- folding and unfolding code blocks,
- changing compression ratio via the slider and the text field next to it, by using mouse movement and keyboard arrows,
- clicking on the thumbs-up icon to give feedback.

Changes to the corresponding system components were made iteratively throughout this process.

Similarly, the application's response to incorrect values was tested:

- For compression ratio: A non-numeric or out-of range value given via the text-field is edited to either 0, 50, or 100 by JavaScript. A non-numeric value given via URL results in a 404, an out-of-range value given via URL results in compression ratio 100.
- A non-existent file or project given via URL results in a 404.

These tests were done in Mozilla Firefox versions 34.0.5 to 36.0 and Opera 26 to 28 under Linux. Testing under a wider range of browsers/versions could not be achieved at this stage, as the application was not hosted on the web (as previously discussed in section 3.7).

Chapter 5

Conclusion

This report described the design, implementation and evaluation of the web application developed in this project. As a user interface to the Machine Learning for Analysis of Software Text tool, it is able to automatically summarise source code by folding away blocks which are not essential to understanding it. Users are able to review source code at different levels of detail by adjusting the compression rate at which it is summarised. They can also browse files from the same project or navigate to other GitHub projects.

This project focused on usability by designing and implementing a simple graphical user interface, and providing a convenient URL-rewriting feature which allows users to navigate from a GitHub file to its summarised version via minimal URL substitution. The bases for extra features such as user feedback and additional browser functionality were developed, and some suggestions for future work were outlined in the report.

The system was developed in independent modules, which aided implementation and evaluation, and also added functionality by allowing the user to only view the source code they are interested in, or interact with it through a project browser. Evaluation showed that the application followed the expected behaviour upon user interaction, produced correct code as tested on a few popular GitHub repositories, and exhibited no prohibitive concerns related to response time or memory requirements.

The code for this application was written with the effort to provide a modular and maintainable environment for future work on the project. Its configuration was set up to allow its portability to a production server.

Bibliography

[1] Ace Editor Google Group. https://groups.google.com/forum/#!forum/ace-discuss.

[2] J Angerpikk. Most Popular Application Servers in 2014. https://plumbr.eu/blog/most-popular-application-servers-in-2014, 2014. [fetched 28/03/2015].

[3] caniuse.com. Range Input type browser compatibility. http://caniuse.com/#search=slider, 2006. [fetched 31/03/2015].

[4] E Jendrock et al. Chapter 4: JavaServer Faces Technology, The Java EE Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/bnaph.html, 2013. [fetched 01/04/2015].

[5] E Jendrock et al. Chapter 5: Introduction to Facelts, The Java EE Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/giepx.html, 2013. [fetched 29/03/2015].

[6] E Jendrock et al. JavaServer Faces Technology Benefits, The Java EE Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/bnapj.html, 2013. [fetched 01/04/2015].

[7] E Jendrock et al. The Java EE 6 Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/giepx.html, 2013. [fetched 01/04/2015].

[8] github.com. Edit like an Ace. https://github.com/blog/905-edit-like-an-ace, 2011. [fetched 22/01/2015].

[9] Tim Holloway. JSF vs JSP + Servlets, coderanch.com. http://www.coderanch.com/t/644329/JSF/java/JSF-JSP-Servlets, 2015. [fetched 01/04/2015].

[10] JavaServer Pages Implementation. Source Control. https://java.net/projects/jsp/sources/svn/show/, 2014. [fetched 28/03/2015].

[11] Q Mahmoud. Java Technologies for Web Applications. http://www.oracle.com/technetwork/articles/java/servlets-jsp-140445.html, 2003. [fetched 29/03/2015].

[12] D Nourie. Java Technologies for Web Applications. `http://www.oracle.com/technetwork/articles/java/webapps-1-138794.html`, 2006. [fetched 29/03/2015].

[13] stackoverflow.com. Programmatically fold code in ACE editor. `http://stackoverflow.com/questions/12823456/programmatically-fold-code-in-ace-editor`. [fetched 22/01/2015].

[14] J Fowkes R Ranca M Allamanis M Lapata C Sutton. Autofolding for source code summarization, 2014.

[15] tomcat.apache.org. Apache Tomcat Versions. `http://tomcat.apache.org/whichversion.html`. [fetched 29/03/2015].

[16] tutorialspoint.com. JSF Overview. `http://www.tutorialspoint.com/jsf/jsf_overview.htm`. [fetched 01/04/2015].

[17] wikipedia.org. Comparison of Web Application Frameworks: Java. `http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java`. [fetched 28/03/2015].