Assignment 2 - GUI Implementation and Testing

Student Name: **Joel Pillar-Rogers** Student Id: **2017545** Student FAN: **pill0032**

This document describes the creation of a GUI and JUnit tests for the YouTubeTrender application built in Assignment 1. It is divided into four phases: parser, sorter, indexer and extensions.

Phase 1 (parser GUI and JUnit testing):

In this first phase I implemented a basic GUI for my YouTubeTrender application and wrote JUnit tests for the Parser code. I also added a Dialog Box for when exceptions are thrown during parsing.

Basic GUI implementation

As a first step, I copied my own YouTubeTrender application over the one provided for the assignment. I compared the two applications and found they were very similar in both internal logic and output, and my version was more feature rich. I just had to edit a few variable names to make everything work.

I created a DefaultListModel in the Frame class and connected it to the component JListVideo using the *setModel()* method (see **Figure 1**). I then populated the model from the List generated by the YouTubeDataParser with a simple 'for' loop and *addElement()* method.

```
DefaultListModel<YouTubeVideo> model = new DefaultListModel<>();

// connects the data/model to the view
this.jListVideo.setModel(model);

// populates the model from the ytParser
for (int i = 0; i < list.size(); i++) {
    model.addElement(list.get(i));
}
```

Figure 1: Model/GUI connection

Next, I used the built in Listener method *jListVideoValueChanged()* to link the text fields on the GUI to the model so that they update when a ListSelectionEvent is sent. I took an index number from the JUnitList to find the correct video in the model and sent that back to the components to display (see **Figure 2**). I did it this way to add separation between the model and view/controller.

```
private void jListVideoValueChanged(javax.swing.event.ListSelectionEvent evt) {
    if (!jListVideo.getValueIsAdjusting()) {
        int index = this.jListVideo.getSelectedIndex();
        this.jTextFieldChannel.setText(model.get(index).getChannelTitle());
        this.jTextFieldDate.setText(model.get(index).getPublishedAt());
        this.jTextFieldTitle.setText(model.get(index).getTitle());
        this.jTextFieldViewCount.setText(String.valueOf(model.get(index).getViewCount()));
        this.jTextAreaDescription.setText(model.get(index).getDescription());
    }
}
```

Figure 2: Link jTextFields to the Model

Below is the first implementation of the GUI (**Figure 3**).

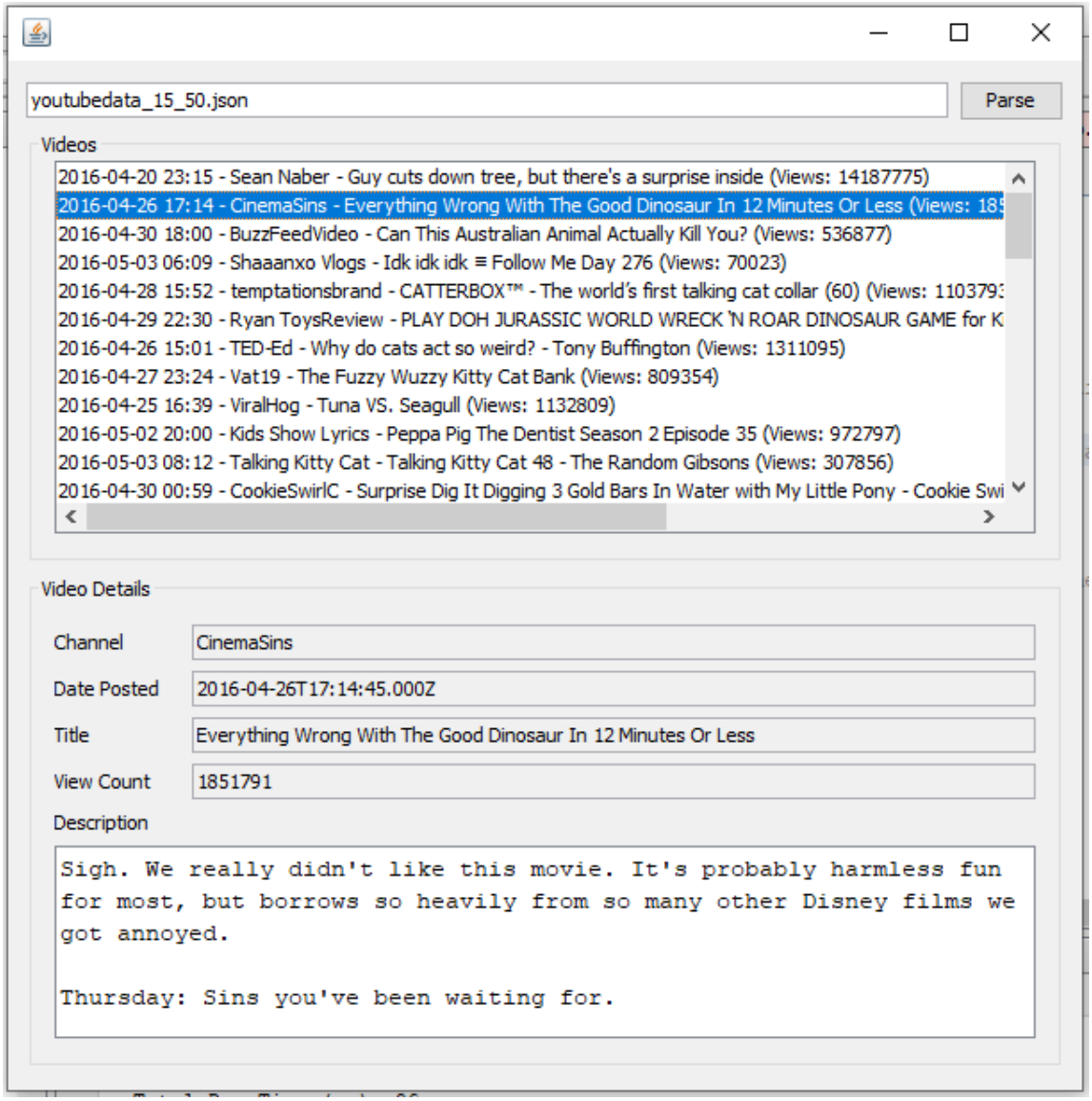


Figure 3: GUI for Phase 1

Exception message dialog boxes

I added a Dialog Box to display any of the exceptions thrown by YouTubeDataParser. I used the *getMessage()* method of the YouTubeDataParserException and passed it to the *showMessageDialog()* static method of the JOptionPane class (see **Figures 4** and **5**).

```
try {
    // Run the YouTubeParser over the file in the data directory
    ytParser.parse("data" + File.separator + dataFile);
    // Get the ArrayList from the ytParser, and save as a List.
    list = ytParser.getArray();
    // Catch any of the parsing exceptions
} catch (YouTubeDataParserException ex) {
    // Prints out a DialogPane with the relevant error message
    JOptionPane.showMessageDialog(null,ex.getMessage());
}
```

Figure 4: Code to throw Exception Dialog Box

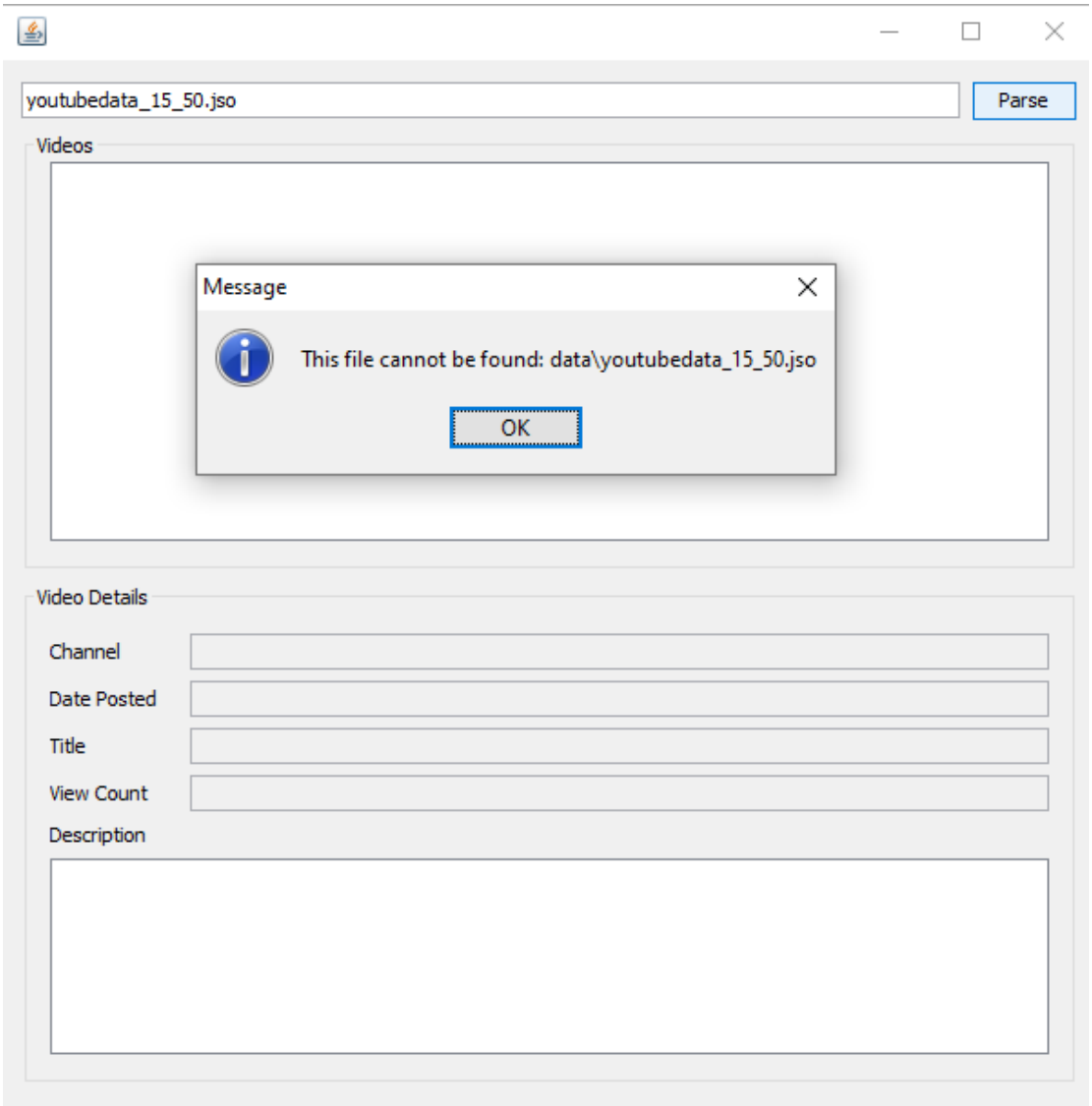


Figure 5: DialogBox to display Exception messages.

Testing of Phase 1 classes

I implemented JUnit tests for Phase 1 classes, which include YouTubeVideo, YouTubeDataParser and YouTubeDataParserException classes, although I left testing of the Sorting methods of YouTubeDataParser until Phase 2. There were a lot of getter and setter methods to test so I kept these tests quite simple and gave more attention to testing the overall parsing process. I avoided testing *System.out.print()* methods, particularly if these were very long strings of formatted text, because they had no real impact on the functioning of the program.

I didn't test the Exception class directly because it will be tested indirectly in YouTubeParserTest. Following Trent's advice, I also didn't attempt to test the YouTubeTrenderFrame class either.

This left the following two test classes to be implemented for this phase:

- YouTubeVideoTest
- YouTubeDataParserTest

To begin, I created these two Test classes and ran them immediately. As expected, they all failed. As I implemented the tests, one by one they started to pass (**Figure 6**).

```
/**
 * Test of parse method, of class YouTubeDataParser.
 */
@Test
public void testParse1() {
    System.out.println("parse1");
    YouTubeDataParser ytp = new YouTubeDataParser();
    try {ytp.parse("data/youtubedata_15_50.json");
    } catch (YouTubeDataParserException ytdpe) {}
    int expResult = 50;
    int result = ytp.size();
    assertEquals(expResult, result);}
```

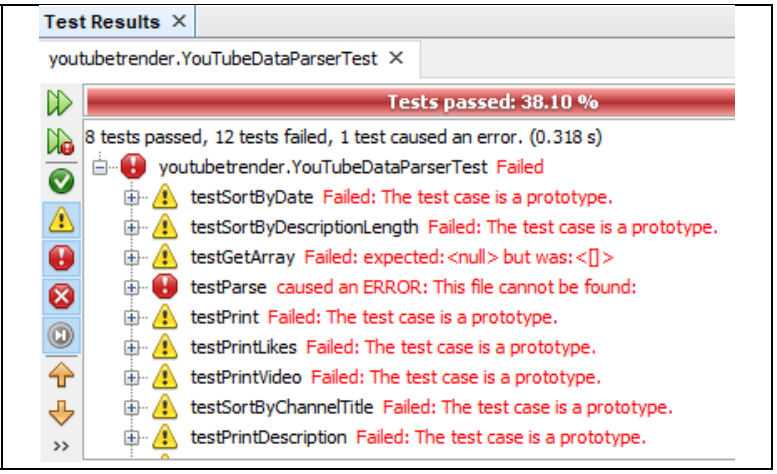


Figure 6: Some test code implementation and results

More complex testing was required in the YouTubeDataParserTest class. I first tested *getSize()* and *getVideo()* by running the parser over several of the provided data files to make sure they would return the correct results. I also tested that I would get the expected error for out-of-range videos with getVideo().

I then tested the *parse()* method directly. I designed tests for these instances:

- completes and contains the correct number of elements
- throws expected exception on incorrect filename
- throws expected exception on corrupted file
- throws expected exception when trying to read null fields

Below is a code snippet of a test for corrupted data (**Figure 6**):

```
/**
 * Test of parse method for corrupted data, of class YouTubeDataParser.
 */
@Test(expected = YouTubeDataParserException.class)
public void testParse2() throws Exception {
    System.out.println("parse2");
    YouTubeDataParser ytp = new YouTubeDataParser();
    String expResult = "Corrupted Json File: ";
    try {
        ytp.parse("data/youtubedata_malformed.json");
    } catch (YouTubeDataParserException y) {
        System.out.println(y.getMessage());
        if (y.getMessage().contains(expResult)) {
            throw new YouTubeDataParserException("");
        }
    }
}
```

Figure 6: Test code for exceptions in the *parse()* method

Finally, I created a TrenderTestSuite to run all my class tests at once.

Phase 2 (sorter GUI and JUnit testing):

In this second phase, I added sorter radio buttons to the GUI, and ran JUnit tests on the sorter methods.

GUI Sorting

I thought about grouping the radio buttons into one action listener command, but NetBeans was struggling with this and each button action would also be calling different sort methods. Eventually, I decided to make a separate listener for each radio button. NetBeans still did much of the heavy lifting, registering the action listener and sending the action event to the required method (see **Figure 7**).

```
buttonGroup1.add(jRadioButtonChannel);
jRadioButtonChannel.setText("Channel");
jRadioButtonChannel.setToolTipText("sort by channel title");
jRadioButtonChannel.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jRadioButtonChannelActionPerformed(evt);
    }
});
```

Figure 7: NetBeans generated code to register a RadioButton listener

For each listener method, I used the YouTubeDataParser methods to sort the list before repopulating the model. I did this so that I could use the methods I had already created and trusted from Assignment 1. I also made a private *repopModel()* function to avoid rewriting this code many times. An example of both of these methods is below (see **Figure 8**).

```
 * Sorts the model by the Description Length of each video.
 */
private void jRadioButtonDescriptionActionPerformed(java.awt.event.ActionEvent evt) {
    if (list != null) {
        ytParser.sortByDescriptionLength();
        list = ytParser.getArray();
        repopModel();
    }
}
/**
 * Re-populates the model from the updated list
 */
private void repopModel() {
    for (int i = 0; i < list.size(); i++) {
        model.set(i, list.get(i));
    }
}
```

Figure 8: Sorting and Re-populating code

Below is the second implementation of the GUI (**Figure 9**).

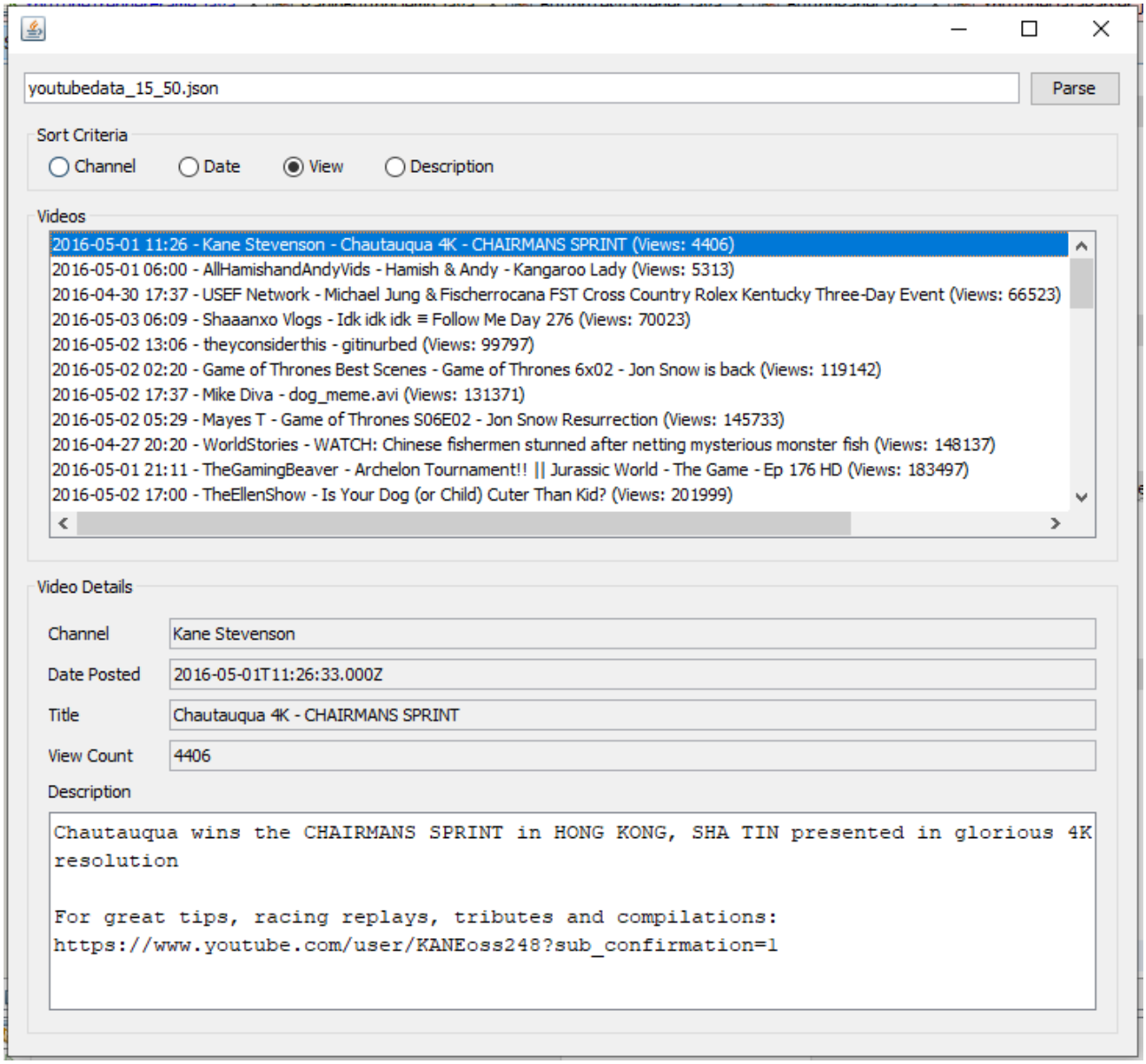


Figure 9: GUI for Phase 2

Testing of Phase 2 classes

Testing in this phase was all about the sorters. Again, I didn't test the Comparators directly, but indirectly through their effect on the *sortbyX()* methods. These methods were all part of the YouTubeDataParser class. I had 8 sort methods to test from Assignment 1:

- sortByDate()
- sortByTitle()
- sortByChannelTitle()
- sortByViews()
- sortByDescriptionLength()
- sortByLikes()
- sortByLikeRatio()
- sortByComments()

For each one, I ran the sort method and concatenated the titles of the first and last video into a single string. I then checked this string against the values provided directly by the JSON data files.

I couldn't produce any exceptions using the sort methods, probably because they are based on the very stable *Collections.sort.* I did test sorting an empty array, but this didn't produce any exceptions, which could be considered a test itself.

At this point I couldn't think of any other tests to run. I decided to try another sort order test, where I looped through every element of a sorted list to make sure every pair of videos was ordered correctly. Below is an example of this code (**Figure 10**).

```
/**
 * Test of sortByDate method for each item, of class YouTubeDataParser.
 */
@Test
public void testSortByDate3() throws Exception {
    System.out.println("sortByDate3");
    boolean sorted = true;
    YouTubeDataParser ytp = new YouTubeDataParser();
    ytp.parse("data/youtubedata_loremipsum.json");
    ytp.sortByDate();
    for (int i = 0; i < ytp.size() - 1; i++) {
        sorted = ytp.getVideo(i).getPublishedAt().compareTo(ytp.getVideo(i + 1).getPublishedAt()) < 0;
    }
    assertTrue(sorted);
}
```

Figure 10: Testing the sort code

Running the TrenderTestSuite at this point produced **47** successful tests.

Phase 3 (indexer GUI and JUnit testing):

In this third phase I added GUI functionality for indexing videos and appropriate JUnit tests to determine their accuracy.

GUI Indexing

I began by adding the Trending pane to the GUI and asking Netbeans for a listener for when JButtonIndex is pressed. The method in this listener takes the existing ytParser list and indexes each word into a WordItem list, then displays them in the new Trending pane. There is a new linked model underlying the Trending pane. This code is below (see **Figure 11**).

There's more to do here, though. The image in the Assignment instructions appears to show that selecting an indexed WordItem will display the videos associated with it. This requires another listener method for when a WordItem is selected, which should in turn go back to the model underlying the JListVideo pane and display only the related videos.

```
        try {
            // Run the YouTubeIndexer over the ytParser object
            ytIndexer.index(ytParser);
            // Get the ArrayList from the ytIndexer, and save as a List.
            trendList = ytIndexer.getSortedWordItems();
            // Catch any of the indexing exceptions
        } catch (YouTubeDataParserException ex) {
            // Prints out a DialogPane with the relevant error message
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
    }
    if (trendList != null) {
        // ytParser data has been indexed successfully!
        // connects the data/model to the view
        this.jListTrending.setModel(trendModel);

        // populates the model from the trendList obtained from ytIndexer
        for (int i = 0; i < list.size(); i++) {
            trendModel.addElement(trendList.get(i));
        }
```

Figure 11: Setting up the Trending model and JList

This part was tricky. I had a lot of trouble switching from WordItems to Video items and back and sorting the reduced WordItem list of videos. Eventually, I created a Set of videos that existed alongside the model. Whenever a selection was made anywhere, it could either update this Set (e.g. when selecting aWordItem), or only display videos that were in this Set (e.g. when sorting the YouTubeItems). I also added a Boolean *modelOp,* to stop the JListVideo field trying to update while another operation was ongoing. Some of this code is below (**Figure 12** and **13**).

```
private void jListTrendingValueChanged(javax.swing.event.ListSelectionEvent evt) {
    if (!jListTrending.getValueIsAdjusting()) {
        int index = jListTrending.getSelectedIndex();
        vidSet = trendList.get(index).getPosts();
        modPop();
    }
}
```

Figure 12: Listener code to update the Set when selecting a WordItem

```
private void modPop() {
    modelOp = true;
    model.clear();
    for (int i = 0; i < ytParser.size(); i++) {
        if (vidSet.contains(ytParser.getVideo(i))) {
            model.addElement(ytParser.getVideo(i));
        }
    }
    modelOp = false;
}
```

Figure 13: Helper method to only update the model with Set videos

Below is the third implementation of the GUI (**Figure 14**).

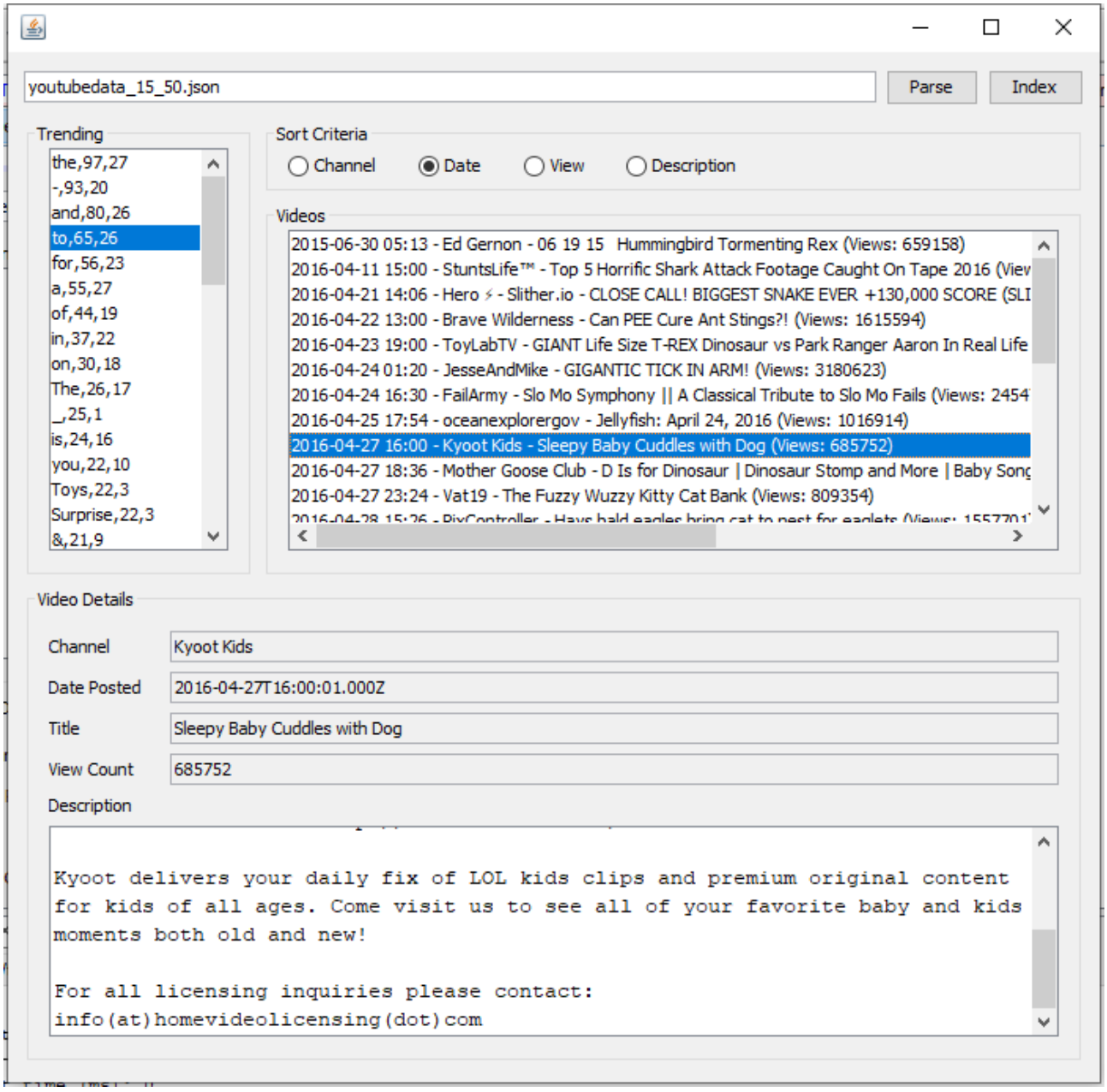


Figure 14: GUI for Phase 3

Testing of Phase 3 classes

I used this testing phase to test the operation of my indexer classes. There are two main classes to test.

- YouTubeWordItemTest
- YouTubeVideoIndexerTest

The YouTubeWordItem class had mostly getters and setters, so I wrote simple code to quickly verify them.

To test the YouTubeVideoIndexer class, I began by testing whether the *index()* and *countUniqueWords()* methods were returning the right number of words (given in the JSON files).

Next, I tested the *getSortedWordItems()* method, which returns a sorted ArrayList of all the WordItems. I ran this on a data file and tested whether the first and last WordItems in the ArrayList were the most and least used words according to the JSON info file (see **Figure 15**).

I also tested whether the IndexOutOfBoundsException would be thrown if the method was called on an empty Set.

```
 * Test of getSortedWordItems method, of class YouTubeVideoIndexer.
 */
@Test
public void testGetSortedWordItems() throws Exception {
    System.out.println("getSortedWordItems");
    YouTubeVideoIndexer yti = new YouTubeVideoIndexer();
    yti.index(parser1);
    String result = yti.getSortedWordItems().get(0).getWord();
    result += yti.getSortedWordItems().get(yti.getSortedWordItems().size() - 1).getWord();
    String expResult = "theJimbo";
    assertEquals(expResult, result);
}
```

Figure 15: Test code for the sorted Indexer

I tested the *getMostUsedWord()* method. I ran the method and tested whether it was the same word as in the JSON info file. I also tested whether it threw the right exception if run on an empty set.

Finally, I tested the *getWordItem()* method. I ran the indexer and submitted a word string to be found in the Indexer's HashMap. This returned a WordItem which I tested for accuracy. I also tested that running this method on an absent word would return null.

At this point I also thought about testing some of the YouTubeTrenderFrame methods. However, as I scanned the code I realised that most of the methods were built on these other methods I'd already tested. In addition, they were all private methods, so wouldn't come up in the JUnit tests (I'm sure there is a way round this).

In total, running the TrenderTestSuite at the end of Phase 3 produced **63** successful tests.

Phase 4 (GUI extensions):

In Phase 4, I added additional functionality to the GUI. Specifically, the GUI now has these features:

1. Additional data fields and sort methods
2. All fields clear on new Parse; all video data fields clear on new Index
3. Mnemonics for keyboard commands and tooltips everywhere
4. Can choose files with the browse button

Additional data fields and sort options

My application code from Assignment 1 collected additional data fields than were required, including:

- "categoryId"
- "channelId"
- The metadata ("kind", "etag", "ID')
- The statistics ("likes", "dislikes", "comments", "favorites")

It also had additional sort methods (these were tested in Phase 2), including:

- *sortByChannelTitle()*
- *sortByLikes()*
- *sortByLikeRatio()* (this sorts by the ratio of likes to dislikes)
- *sortByComments()*

The Category ID fields could be used to sort into categories, but I haven't implemented this because I don't know what each category number represents. The Channel ID and video metadata are largely meaningless for this application given I can sort by Title or Channel Title. I also found the "favorites" field was zero for each video (I don't recall a "favorites" button in YouTube).

I think the other statistics are interesting to view and sort by. I changed the GUI to accommodate these data fields and sort methods and implemented the underlying code. I won't paste any of this code here because it's the same as previous code, with just the variables changed.

Clear all fields on Parse and all data fields on Index

To make the program look cleaner and avoid confusing situations, I coded a *clearGUI()* method to clear the text fields, the indexer model and the radio button selection on each Parse (**Figure 16**).

```
private void clearGUI() {
    modelOp = true;
    //clear models
    model.clear();
    trendModel.clear();
    //clear radio button selections
    buttonGroup1.clearSelection();
    // clear text fields
    jTextFieldChannel.setText("");
    jTextFieldDate.setText("");
    jTextFieldTitle.setText("");
```

Figure 16: Code to clear the GUI

Tooltips and allow keyboard commands

I added tooltips to each button that could be interacted with. This gives the user more information about what a button does before they press it. I also added tooltips to the text fields to provide some more information about what is being displayed.

I added mnemonics to each of the action buttons so they could be accessed via the keyboard. I found that mnemonic underlines weren't always displayed by default (I think this is a Windows setting) so I added code in the *main()* method to force them to always display (**Figure 16**).

```
// Set System to always show mnemonic underlines
UIManager.getLookAndFeelDefaults().put("Button.showMnemonics", true);
```

Figure 16: Code to ensure mnemonic underlines appear

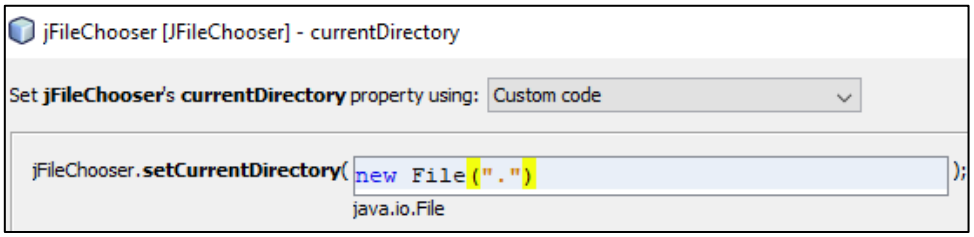
Browse folders for JSON files to Parse

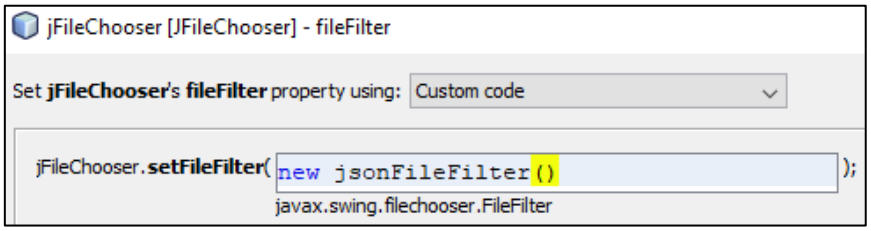
I added a FileChooser so that a user can browse their folders to choose a new JSON file to parse, rather than having to type it in. I added a FileChooser to the Navigator from the SwingMenu. I then added a "Browse" button and linked it to the FileChooser with the code below (**Figure 17**).

```
private void jButtonBrowseActionPerformed(java.awt.event.ActionEvent evt) {
    int returnVal = jFileChooser.showOpenDialog(this);
    if (returnVal == jFileChooser.APPROVE_OPTION) {
        File file = jFileChooser.getSelectedFile();
        jTextFieldDataFile.setText(file.getName());
        clearGUI();
    } else {
        System.out.println("File access cancelled by user.");
    }
}
```

Figure 17: Code for linking the FileChooser and GUI

I added two other features to the FileChooser. I added an inner class that tells the FileChooser only to look for ".json" files. I also changed the default directory to always point to the location of the YouTubeTrender project (see **Figure 18**).






```
/**
 * Inner class to override the default file filter to only show JSON files
 */
class jsonFileFilter extends javax.swing.filechooser.FileFilter {

    @Override
    public boolean accept(File file) {
        // Allow only directories, or files with ".json" extension
        return file.isDirectory() || file.getAbsolutePath().endsWith(".json");
    }

    @Override
    public String getDescription() {
        // This description will be displayed in the dialog box
        return "JSON files (*.json)";
    }
}
```

Figure 18: Code for setting the default directory and allowed files

I tried hard to embed a YouTube video player on the GUI, but it proved beyond me for now. I'd be interested to see how someone implemented this before.

Therefore, the final GUI is below (**Figure 19**).

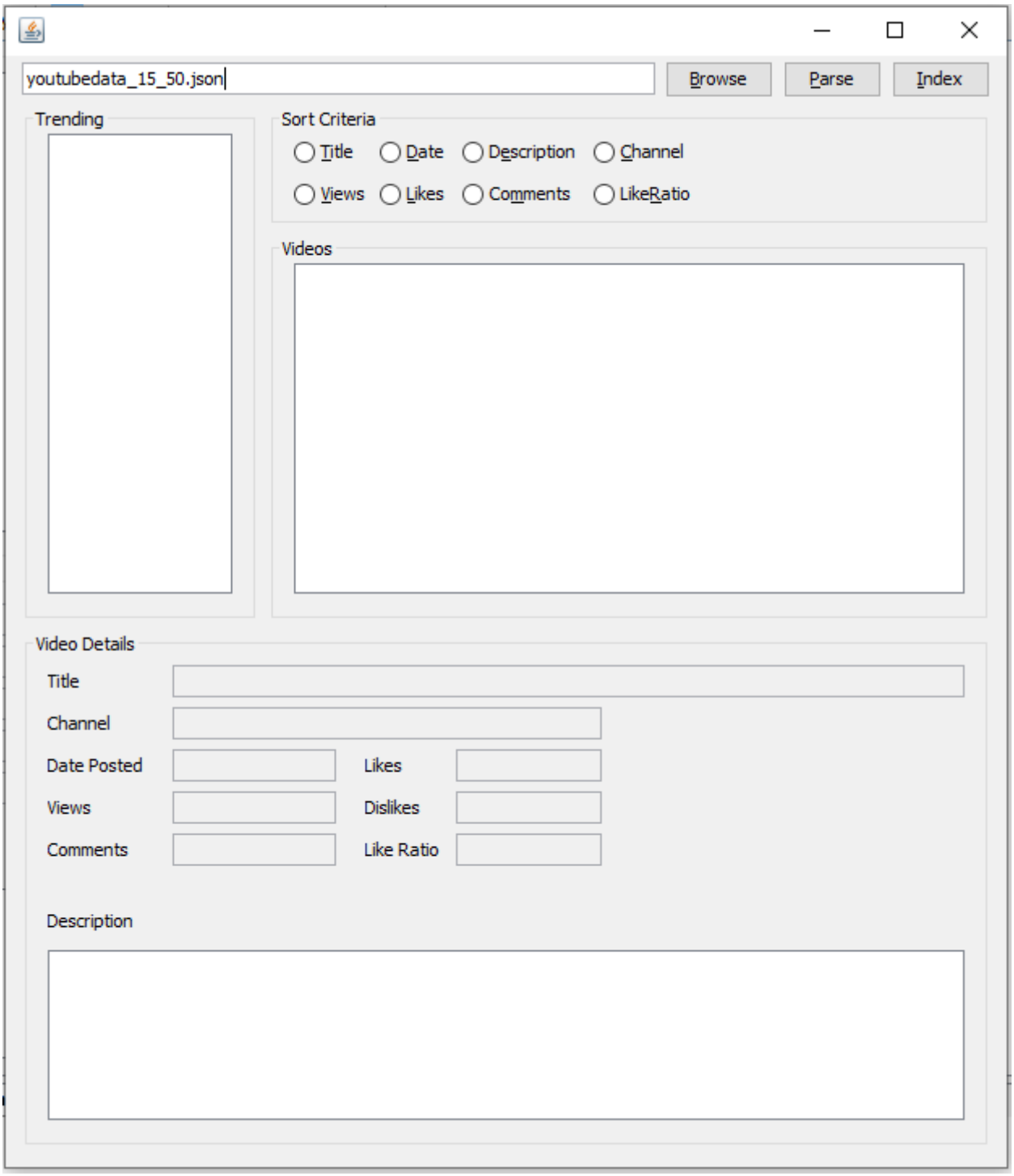


Figure 19: The Final GUI

Final revision number: **384**.