

Development Stage

HND Graded Unit

Colin J. Barr

Contents

1	Production	2
1.1	Problem domain	2
1.1.1	Visitor pattern	2
1.1.2	Migrations	3
1.1.3	Serializer	5
1.1.4	Multithreading	6
1.1.5	Audit Report Output	7
1.2	UI domain	8
1.2.1	Layout	8
1.2.2	Navigation	10
1.2.3	Graph view	10
1.2.4	Localisation	11
1.2.5	Gutter Audit Tooltip	12
1.2.6	Drag & Drop Opening	13
1.3	Unfamiliar libraries	13
1.3.1	JavaParser	13
1.3.2	GSON	13
1.3.3	WebLaf	14
1.3.4	j2html	14
1.3.5	RSyntaxTextArea	14
1.3.6	Dependency Tree	14
1.4	Error handling	15
1.4.1	GitHub URL verifier	15
1.4.2	GitHub directory validation	16
1.4.3	Parse problem view	16
1.4.4	Checksum warning	17
1.4.5	Custom Exception	18
1.5	Internal documentation	19
1.6	Version control	21

2	Testing	23
2.1	Cross platform	23
2.2	Profiling	24
2.2.1	Heap Overview	24
2.2.2	Largest Objects	25
2.2.3	Object Querying	27
2.2.4	Querying NodeList	29
2.3	Functional testing	31
2.3.1	Testing of Invalid Source Files	31
2.3.2	Testing of Differing Checksums	33
2.4	Migrations testing	33
2.5	Test plan	35
2.5.1	Unit Test: GitHub Verifier	35
2.5.2	Unit Test: Audit Model Test	36
2.5.3	Unit Test: MethodTreeVisitor	38
3	Documentation	41
3.1	Online help features	41
3.1.1	FAQ Website	41
3.2	This Document	42

1 Production

1.1 Problem domain

1.1.1 Visitor pattern

jAudit utilises a visitor pattern to traverse the AST in order to yield nodes from it. Most notably, the *MethodTreeVisitor* class visits *MethodDeclaration* nodes and appends them to the root - class - node.

```

1  /**
2   * Method-visitor designed for appendage to method-tree
3   */
4  public class MethodTreeVisitor extends VoidVisitorAdapter<ClassTreeNode> {
5
6      /**
7       * Visit AST method
8       * @param methodDecl method declaration node
9       * @param root tree node being appended to (where the methods are leaves)
10      */
11      @Override
12      public void visit(MethodDeclaration methodDecl, ClassTreeNode root) {
13          // ensure relevant siblings are traversed by adapter (parent)

```

```

14     super.visit(methodDecl, root);
15     // add method to root node
16     root.add(new MethodTreeNode(methodDecl));
17 }
18
19 }
```

The above visitor is responsible for populating the method tree which used used as a primarily navigational and informative component.

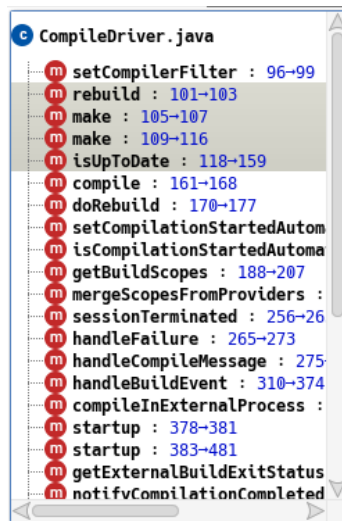


Figure 1: Method tree component populated with visitor

The visitor itself is extending an *adapter* because the *VoidVisitorAdapter* allows subclasses to access its super methods which correctly implement node traversal for all nodes in the AST. Thus, one can isolate specific node(s) as I have done here.

1.1.2 Migrations

jAudit uses a database for efficient storage of audits (and references to the file(s) the audit(s) reference). Since jAudit must create the SQLite tables it requires on the user's machine, I chose to create a system somewhat similar to how Laravel manages database revisions. I opted to create an elementary *migrations* system. Each migration simply contains logic required to *up* (construct) and *down* (reverse) creational aspects of the database. For example, there exists a migration for the creation of both the audit and file table.

The blueprint for a migration can be seen below:

```
1  /**
2   * Migrations allow for the easy creation and destruction of database schemas.
3   */
4  public abstract class Migration {
5      protected DBConnection connection;
6
7      /**
8       * Common constructor for dependency injection of database connection
9       * (see {@link DBConnection}) for easier testing.
10     *
11     * @param connection dependency injectable connection to be used
12     * by the migration
13     */
14     public Migration(DBConnection connection) {
15         this.connection = connection;
16     }
17
18     /**
19     * The method is for creational actions upon the database schema.
20     *
21     * @return whether the migration was completed successfully
22     */
23     public abstract boolean up();
24
25     /**
26     * The method is for reversing the actions of {@link Migration#up()}.
27     *
28     * @return whether the migration's reversal was completed successfully
29     */
30     public abstract boolean down();
31 }
```

As you can see, the Migration itself is an *abstract* class, therefore one can't instantiate it itself. One must instantiate a subclass concretion that implements the abstract methods `up()` and `down()`.

It's also worth noting that the Migration class has a constructor that sets a *protected* `DBConnection`. This is to allow for *dependency injection* so that the migrations can operate on independent connections and be easier to test since we can also supply them with *facade* connections purely for debugging.

The actual dependency-injectable `DBConnection` class is a *singleton* class that acts a minimal wrapper around a `java.sql.Connection`.

1.1.3 Serializer

In order to prepare the context of an audit for storage in a database, the audit context must be tranposed to a JSON representation.

I use Google's JSON library, GSON, to do this. GSON allows you to register an adapter-serializer for custom types. I decided to implement a serializer for the audit context that would consist of the root node (with its content) and all children nodes (without content) and their positions. Only the root node's content is stored because it would be redundant to store extra content already contained within the root node.

The user *reduces* an audit by simply selecting a range of adjacent nodes in a context-trace from the auditor view (shown below).

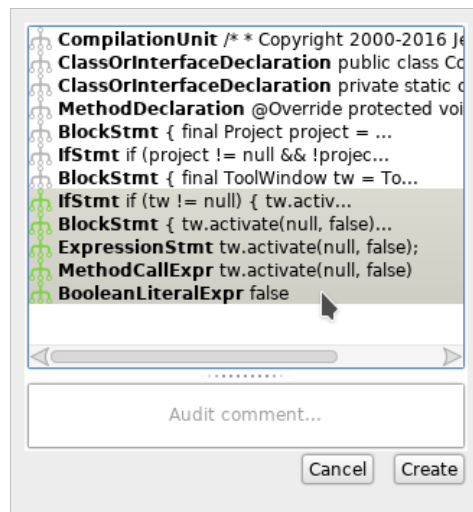


Figure 2: Selecting relevant context via reduction

The above context reduction gets tranposed to:

```

1 {
2   "IfStmt": "if (tw != null) {\n    tw.activate(null, false);\n}",
3   "children": [

```

```

4      ["BlockStmt", [802, 25], [804, 9]],
5      ["ExpressionStmt", [803, 11], [803, 35]],
6      ["MethodCallExpr", [803, 11], [803, 34]],
7      ["BooleanLiteralExpr", [803, 29], [803, 33]]
8  ]
9  }

```

1.1.4 Multithreading

jAudit utilises multi-threading during its parsing stage because the time that parsing each file could potentially take isn't exactly deterministic. Thus, in order to avoid blocking the GUI thread when parsing, it's executed in the background; in another thread.

```

1  private class ParserWorker extends SwingWorker<Boolean, Boolean> {
2
3  @Override
4  protected Boolean doInBackground() throws ParseProblemException {
5      // attempt to parse file
6      parsedFile = initCompilationUnit();
7
8      // if parsing is successful, run method visitor to initialise method-tree
9      if (parsedFile) {
10         initTree();
11     }
12
13     // if parsing failed without throwing ParseProblemException,
14     // return success value anyway
15     return parsedFile;
16 }
17
18 /**
19  * Parsing complete callback, closes progress dialog.
20  */
21 @Override
22 protected void done() {
23     // queue progress-dialog closure in AWT event queue
24     SwingUtilities.invokeLater(() -> showProgressDialog(false));
25
26     // state
27     try {
28         parsedFile = get();

```

```
29         } catch (InterruptedException | ExecutionException e) {  
30             e.printStackTrace();  
31         }  
32     }  
33 }
```

1.1.5 Audit Report Output

Another aspect of the problem domain is the ability to output audit reports from jAudit which outputs reports such as the following:

ImportDeclaration Name

```
import com.jwetherell.algorithms.data_structures.Graph;
```

Don't need this import.

IfStmt BinaryExpr

```
if (project != null && !project.isDisposed() && CompilerTask.showCompilerContent(project, myContent)
    final ToolWindow tw = ToolWindowManager.getInstance(project).getToolWindow(ToolWindowId.MESSAGES)
    if (tw != null) {
        tw.activate(null, false);
    }
} else {
    notification.expire();
}
```

This condition will never be true anyway.

AssignExpr NameExpr

```
arr[k] = v
```

At no point do we check that this index is within bounds of the array.

ExpressionStmt BooleanLiteralExpr

```
tw.activate(null, false);
```

Parameter should be true to signify that thread worker queue is valid.

Figure 3: Example audit report output

1.2 UI domain

1.2.1 Layout

All of jAudit's *views* (GUI) were written by-hand using Java Swing so I was required to pay extra detail to how I laid components out as to retain layout proportions whilst providing a versatile layout.

An example of how I managed to retain proportions is shown below:

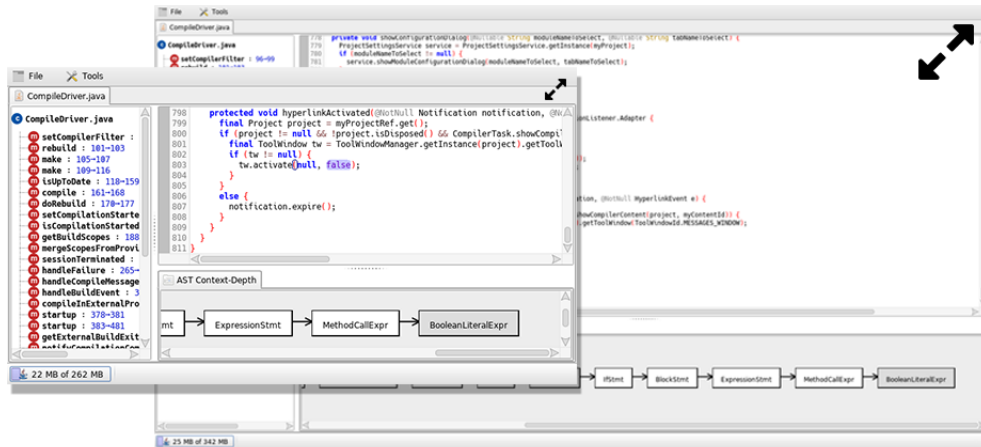


Figure 4: Small and large layout from resizing

To make such a proportional layout possible whilst retaining layout versatility, I opted to wrap the major graphical components of the audit pane into splitpanes. The user can adjust the splits as they like.

On top of this, a visualisation component I created - the graph view - can also be dragged around using mouse drags to increase accessibility as it can be annoying to click precisely on scrollbars. Thus, you can scroll the graph view by simply dragging. That aspect of the graph view is also highlighted below.

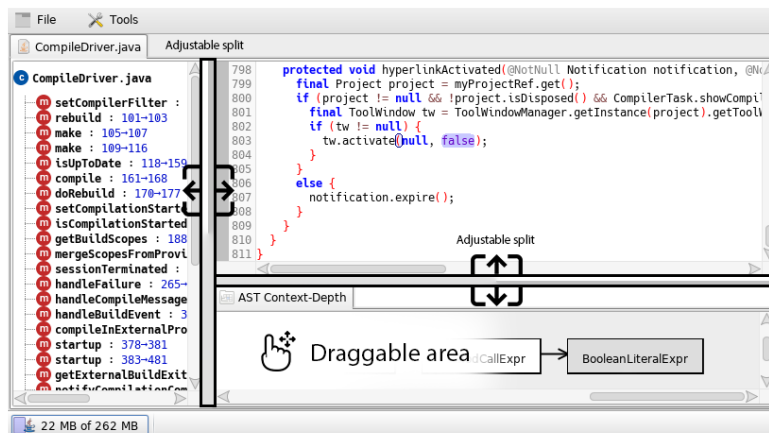


Figure 5: Splits and draggable area highlighted

1.2.2 Navigation

As described in the project planning stage, the method tree is not only a component designed for an overview of the parsed class' method(s), it also is designed to act as a navigational component. By simply selecting any method, the source view will jump to that method's declaration and centre it in the view.

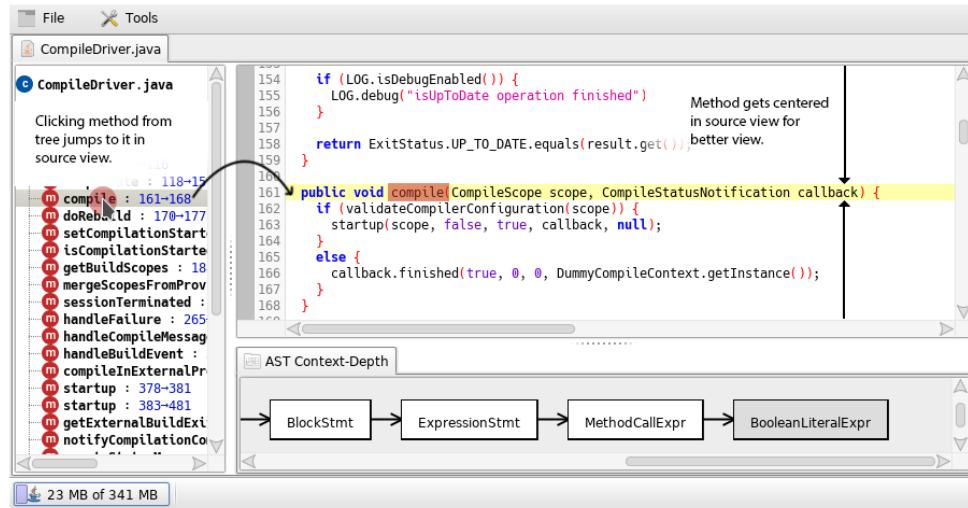


Figure 6: Depiction of on-select navigation and method centering

The method gets centred so that any methods that are navigated to that exist below the current view don't get shown on the last line of the source view as this would require the user to manually scroll down further to actually see the method's implementation.

1.2.3 Graph view

In jAudit's planning stage and proposal, there's lots of elements of visualisation to help the reader understand what the meaning of "AST context-depth", etc. is. I decided that it'd be interesting to create a *custom component* for visualisation purposes. So I opted to create a graph view that displays the AST context-depth as a directed-acyclic graph with an in/out degree or 0 or 1 (sort of like a singly linked list being visualised).

That endeavour resulted in the component below being created:

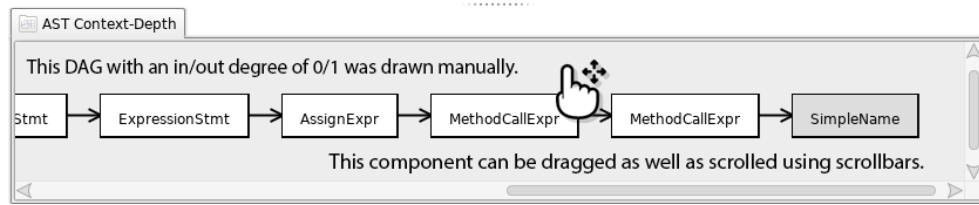


Figure 7: Draggable graph view custom component

It was evident that such a component that could potentially produce graphs that wouldn't be able to be fully visible on smaller screens would require some form of scrolling. So I wrapped it in a scrollable pane and then implemented the ability to click and drag the graph around.

I consider the ability to drag it an aid to both the user experience and accessibility of the program. It feels a lot more fluid to just grab and drag the view rather than manually scroll with the scrollbars (which requires a certain degree of precision when clicking - such a precision that may be taxing for disabled users with a disease such as Parkinson's).

The class hierarchy for this custom component is shown below:

Class Hierarchy

- java.lang.Object
 - java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
 - java.awt.Container
 - javax.swing.JComponent (implements java.io.Serializable)
 - javax.swing.JPanel (implements javax.accessibility.Accessible)

```
org.colin.gui.graph.GraphCanvas<T>
```

```
org.colin.gui.graph.GraphView<T>
  ◦ org.colin.gui.graph.AuditGraph
```

```
org.colin.gui.graph.DrawableNode (implements org.colin.gui.graph.GraphDrawable)
```

Interface Hierarchy

- org.colin.gui.graph.GraphDrawable

Figure 8: Graph class hierarchy

1.2.4 Localisation

jAudit also supports the German language as parts of its localisation support. The choice to support German stemmed from my own recreational learning of German to converse with online friends.

Upon starting jAudit, the user is prompted to choose a language using the dialog depicted below. Of course, the dialog currently only supports (British) English and

(High) German.

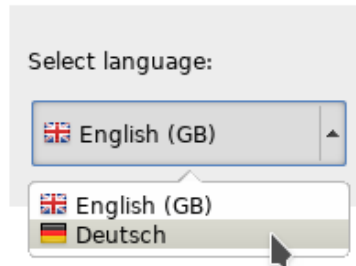


Figure 9: Language selection dialog

An example of the audit view in localised German is shown below:

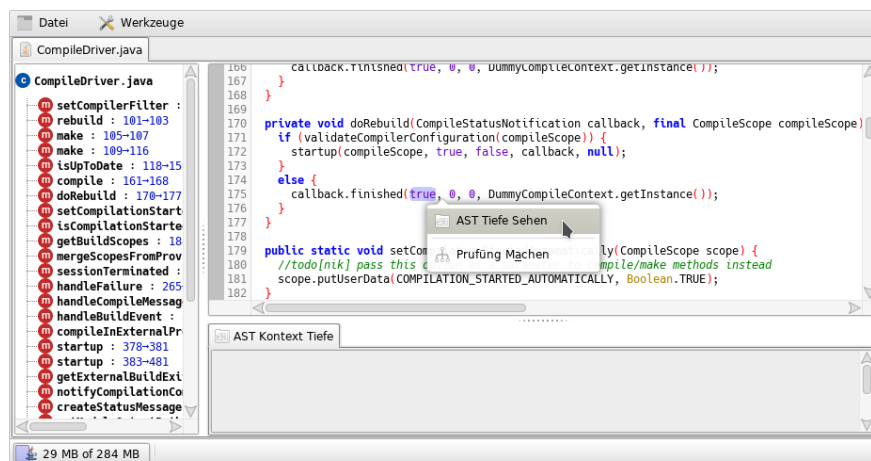


Figure 10: jAudit with German localisation

1.2.5 Gutter Audit Tooltip

When an audit is created (or loaded from a database), its location is bookmarked in the gutter of the text view component displaying the source file. This annotation icon provides the audit comment as its tooltip to allow the user to easily see what's audited at certain source code locations.

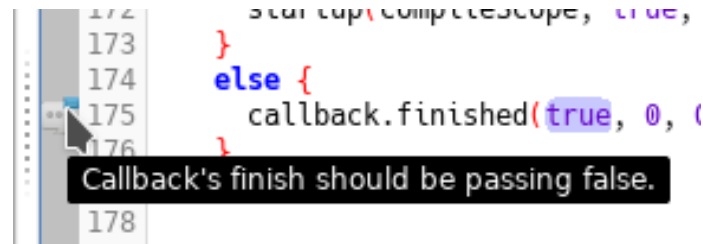


Figure 11: Gutter bookmark with audit comment as tooltip

1.2.6 Drag & Drop Opening

In order to aid the user experience, I implemented the ability to drag & drop files into jAudit's window to open them. I believed this would make opening files somewhat more efficient and practical if the user has a file manager currently open and knows what file they want to open.

1.3 Unfamiliar libraries

Many unfamiliar libraries were used in this project. I used the *Maven* management system to manage dependencies within the project. Such dependencies are listed below.

1.3.1 JavaParser

As noted in the project planning stage, the business logic in jAudit relies heavily on the parsing functionality of JavaParser which is used to construct an AST from given source files. jAudit then traverses this AST to map arbitrary audit comments to reduced ranges within the AST. This allows for construct-relative contextual auditing.

The choice to use *JavaParser* was made just before the project was assigned so using it was actually quite a risk. A few design decisions made in JavaParser caused a few minor hinderences during development. For example, I discovered a few bugs that I've since reported. These were rather obscure bugs such as JavaParser's compilation unit appending newline characters to the end of (this behaviour isn't defined so it caused a few issues with mapping source code to the AST because every selection was, as a result, offset from where it really is in the source file by the number of fields existent in a class). Another nuance is the way JavaParser doesn't really bother with any parsing of comments. This may seem like a good idea since parsers ignore comments (e.g. the entire point of comments), but it actually causes nodes after comments' contents to start with comments which isn't the most useful feature.

1.3.2 GSON

Google's *GSON* project is used to serialize the audit-context to JSON for storage purposes. I considered a hacky approach like Java's own object serialization but

favoured the JSON format for portable practicality, storage, better standardisation, faster parsing, etc.

1.3.3 WebLaf

WebLaF is a LaF (Look and Feel) used for the styling of Java Swing components. I opted to use it because I thought it was visually appealing in comparison to alternatives. It also provides lots of useful tailor-made components (only one of which I used - the memory bar in the status bar).

1.3.4 j2html

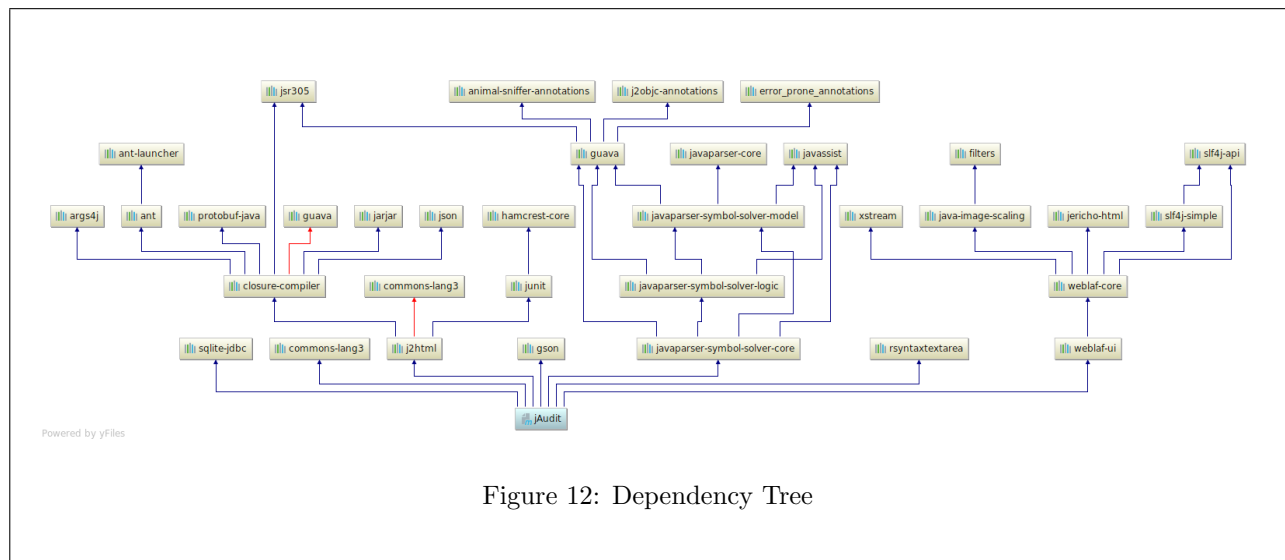
j2html was used in order to make the generation of HTML (for the reports) more idiomatic. The expressive way that you can describe how a HTML document should be rendered (in terms of DOM) improves the maintainability and readability of the code.

1.3.5 RSyntaxTextArea

RSyntaxArea is a custom component I used for the display of syntax-highlighted source code within a text-area. I chose it because implementing syntax-highlighting from scratch would simply result in creating the wheel. There's no practical reason to reinvent the wheel when a mature project such as *RSyntaxArea* exists.

1.3.6 Dependency Tree

The dependency tree for all of these additional libraries can be seen below:



1.4 Error handling

1.4.1 GitHub URL verifier

jAudit supports a feature to open files from GitHub repository URLs. I implemented input validation for this as a first line of defence against invalid input. This involved creating a custom verifier class that verified the *form* of the URL (e.g. the structure - no check that the remote resource exists is performed until the file's download is attempted). It parses it as a URL and makes sure it's in valid form. The verifier itself is a form of **error prevention** because it's not possible to submit the URL until it's confirmed as valid by the verifier (that's bound to the input field). Until the input is valid, the view highlights the input field as red (to signify invalidity) and disables the "Load" button.

This restriction on the GUI as a the form of error prevention can be seen below:

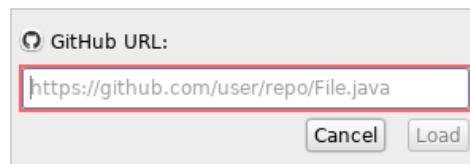


Figure 13: Invalid input with input prompt showing



Figure 14: Invalid input with "Load" button disabled

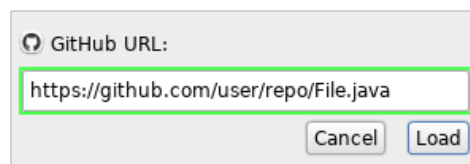


Figure 15: Valid input with enabled "Load" button

A unit test that employs the verifier used by the view above is also documented in the testing section.

1.4.2 GitHub directory validation

When downloading a file from GitHub using the feature described in the previous section, the user is prompted to select a directory to save the file. Input validation was required here because the directory the user selects could potentially be outwith the permissions of the user. For example, on Linux, the default user group doesn't have *write access* to `/usr/lib/`. Attempting to do so would cause a *Permission denied* error. Similarly, an Exception of that nature would be thrown in Java if you attempted to write to a File that references a directory that the user doesn't have permissions for.

I handled the above situation by using a loop that notifies the user of the invalid nature of their directory selection and then continually prompts them to re-input the directory. On top of this, I allowed the user to retain the ability to simply *Cancel* the entire action safely.

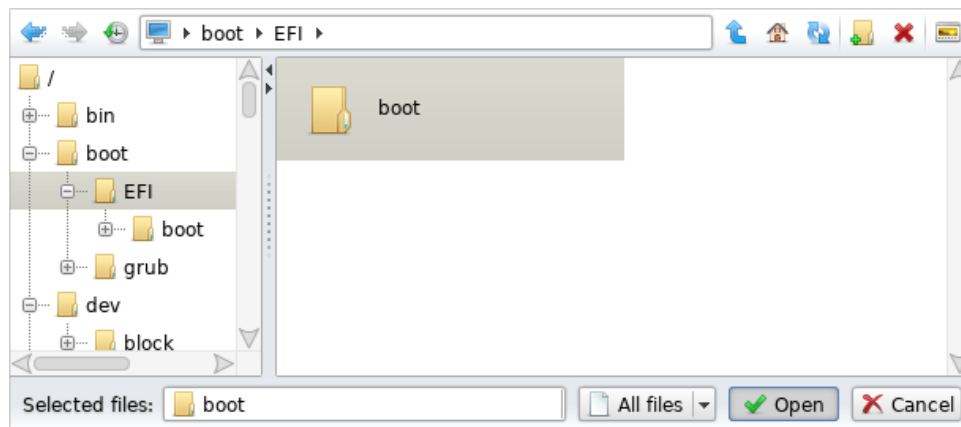


Figure 16: Selection of directory default user has insufficient permissions for (`/boot/EFI/boot`)

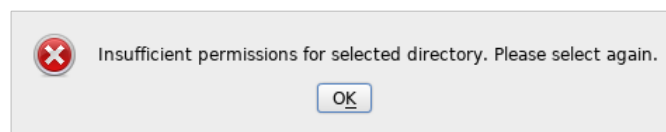


Figure 17: Error prompt shown to notify user of contained error

1.4.3 Parse problem view

In the early stages of the project, any errors encountered during JavaParser's parsing would result in an undescriptive error message being shown. However, after reconsidering the target-demographic of this project, and knowing that I'd personally find

useless information incredibly vexing, I decided to implement a rather rudimentary error view for error resolution.

When a file can't be parsed due to parser errors, the dialog below is shown and the user can optionally choose to view the errors.

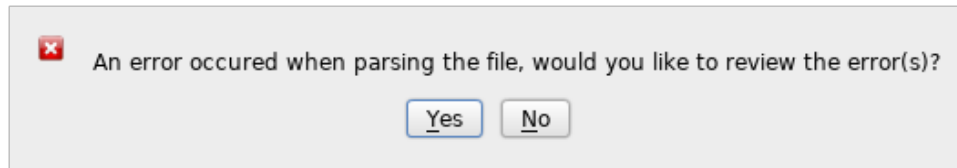


Figure 18: Error message shown when file can't be parsed

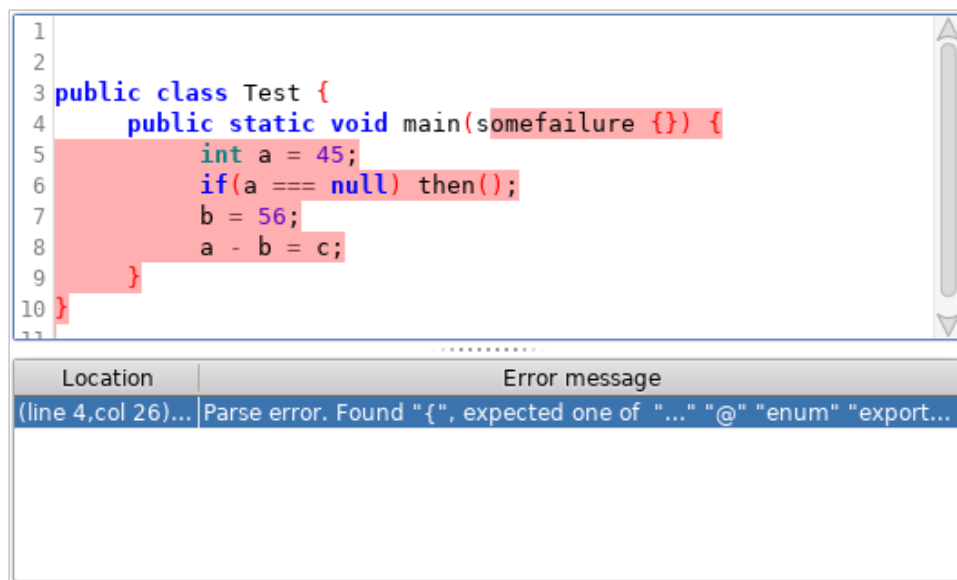


Figure 19: Error prompt shown to notify user of contained error

Granted, the errors highlighted aren't the most helpful but, as noted, jAudit is targeted at expert users (the developers considered skilled enough to audit other peoples' code) so this feature certainly isn't the crux of error resolution within the project.

1.4.4 Checksum warning

In order for jAudit's source-relative audits to retain their source validity, it should be noted that jAudit's files must be treated as immutable once they're inserted.

For this reason, checksums for files are calculated each time they're loaded. If an entry for the loaded file already exists, a checksum comparison is performed. If the checksums don't match, it implies the file has been modified since it was originally inserted into the database. Thus, the user is warned of this.

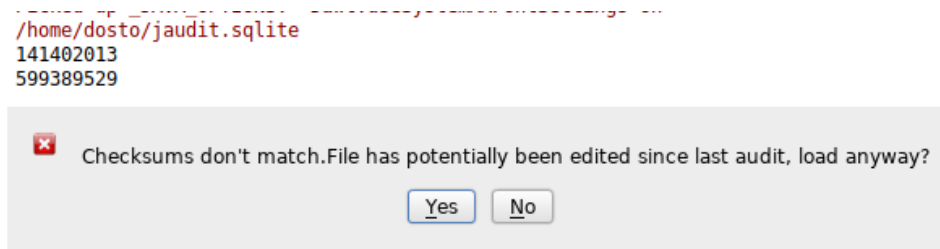


Figure 20: Checksum warning prompt

Above you can see the prompt shown to the user upon the loading of a modified file. jAudit will still attempt to emplace bookmarks for the loaded file with the offsets previously stored. However, any audits that can't be mapped are simply suppressed.

I consider this a lenient form of error prevention as it notifies the user and, given the demographic of the program, the ability to attempt to continue to map audits seems suitable.

1.4.5 Custom Exception

I implemented a custom exception that is thrown when a user attempts to reduce an audit context to an invalid range. A valid range must extend a minimum of two nodes. A range consisting of a single-selected node is invalid.

```

1 package org.colin.exceptions;
2
3 /**
4  * Exception thrown when selected node ranges are invalid.
5  */
6 public class InvalidNodeRangeException extends Exception {
7     /**
8      * Construct invalid node range exception with message
9      * @param message exception message

```

```

10     */
11     public InvalidNodeRangeException(final String message,
12                                     final int first, final int last) {
13         super(message + " " + first + " -> " + last);
14     }
15 }

```

An example where this is implemented is shown below:

```

1  /**
2   * Reduce model's context using sub-listing of an ordered-pair range
3   *
4   * @param first first index
5   * @param last  last index
6   * @throws InvalidNodeRangeException if reduction range is invalid, this is thrown
7   */
8  public void reduceContext(int first, int last) throws InvalidNodeRangeException {
9      if (first == last)
10         throw new InvalidNodeRangeException("Node range must extend
11                                             two nodes", first, last);
12
13         // sublist context
14         context.sublist(first, last);
15 }

```

1.5 Internal documentation

All of jAudit is internally documented using JavaDoc. By taking this approach to documentation, I was able to quickly return to the project after breaks from working on it. JavaDoc also allowed me to easily generated documentation comprising the documentation comments I wrote for everything within the project.

I also spent a few minutes adjusting the default stylesheet provided by JavaDoc to my liking. Examples of such JavaDoc generated output can be seen below:

OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP	
PREV	NEXT
FRAMES	NO FRAMES
Packages	
Package	Description
org.colin.actions	Package containing actions triggered by each view.
org.colin.actions.verifiers	Package containing verifier(s) used for input validation.
org.colin.audit	Package containing classes relating to audits.
org.colin.db	Classes related to database access and global registry.
org.colin.db.migrations	Package containing database actions/migrations.
org.colin.gui	Containing package for every aspect of the Graphical User Interface (MVC classes).
org.colin.gui.controllers	Controllers responsible for business logic behind views.
org.colin.gui.graph	Classes related to the visualisation of the AST context-depth.
org.colin.gui.models	Models used for representing data shown in views and manipulated by controllers.

Figure 21: Extract from index page of documentation

I also attempted to document the package overview to make the resultant documentation a lot easier to navigate.

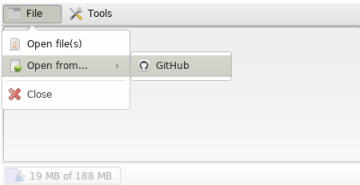
Class Summary

Class	Description
AuditController	Controller used by the AuditView view.
AuditorController	Controller used by the AuditorView view.
LangSelectionController	Controller used by the LangSelectionView view.
MainController	Controller used by the MainView view.
ParseProblemController	Controller used by the ParseProblemView view.
RemoteLoaderController	Controller used by the RemoteLoaderView view.

Package org.colin.gui.controllers Description

Controllers responsible for business logic behind views.

- MainController** - main controller responsible for handling the logic behind the main view. This class also implements `DropTargetListener` to allow users to open files via drag-and-drop.



- AuditController** - controller responsible for handling logic related to the auditing view. Primary method: `audit()`.

Figure 22: Package documentation

As alluded to, the inline documentation was also aided by my choice of IDE that allowed me to practically view documentation without having to navigate to another file or open the documentation in my browser.

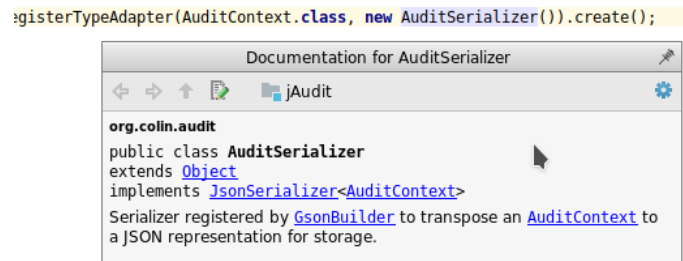


Figure 23: Inline documentation in IntelliJ IDEA

1.6 Version control

In order to keep track of developmental changes within the project, I opted to use the **git** version control system. Git allowed me to effectively manage revisions within the project.

At first, git wasn't used often because I was busy creating the initial beginning of the project so any changes were inherently minor. However, as development progressed and the source code base became harder to manage, git became an invaluable asset to me. I also chose to use the popular online git repository website github to backup my project.

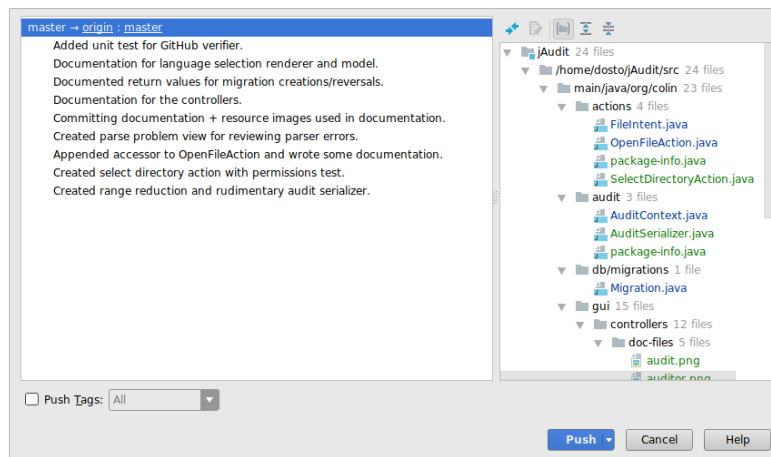
By looking at the git commits, one can see how the project developed over time and when specific features were implemented. This gives anyone intent on committing to the project a better idea of how the project grew over time.

Commits on Apr 29, 2018

Lots of documentation for every package. obsurecolin committed an hour ago	e8553a2	<>
Amended documentation for ColourUtil class. obsurecolin committed an hour ago	b5b2719	<>
Updated Maven dependencies. obsurecolin committed 6 hours ago	d6aacf6	<>
Added unit test for GitHub verifier. obsurecolin committed 6 hours ago	2552019	<>
Documentation for language selection renderer and model. obsurecolin committed 6 hours ago	bb1859c	<>
Documented return values for migration creations/reversals. obsurecolin committed 6 hours ago	76b89a1	<>
Documentation for the controllers. obsurecolin committed 6 hours ago	7b800c2	<>
Committing documentation + resource images used in documentation. obsurecolin committed 6 hours ago	ee8a733	<>
Created parse problem view for reviewing parser errors. obsurecolin committed 6 hours ago	65a0c7d	<>

Figure 24: List of commits

Luckily, the IDE I used for the development of jAudit - IntelliJ IDEA - supported git integration out-of-the-box. This made using git to manage the project correctly incredibly practical and easy.

Figure 25: IntelliJ IDEA's *git* Feature

2 Testing

2.1 Cross platform

Since jAudit is written in Java, it should be able to run on all major operating systems such as Windows, Mac, and mainstream Linux distributions.

I tested the functionality of jAudit on both Linux & Windows.

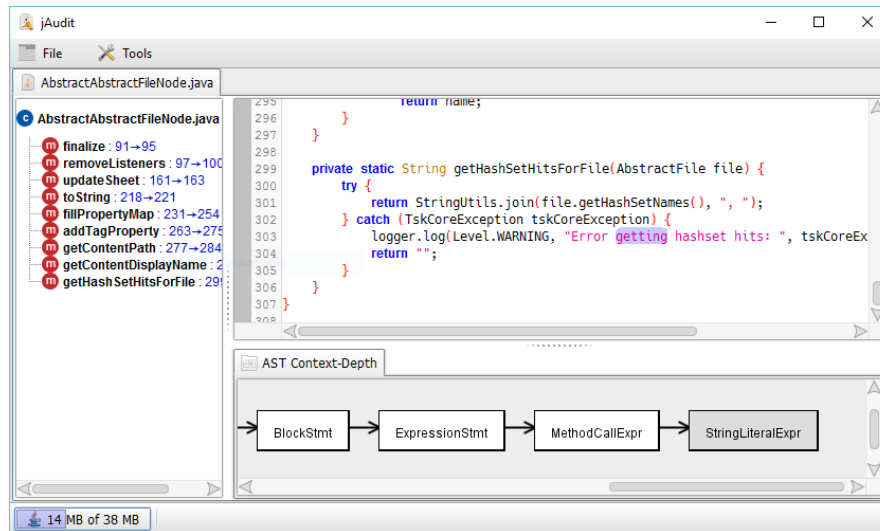


Figure 26: jAudit running under Windows

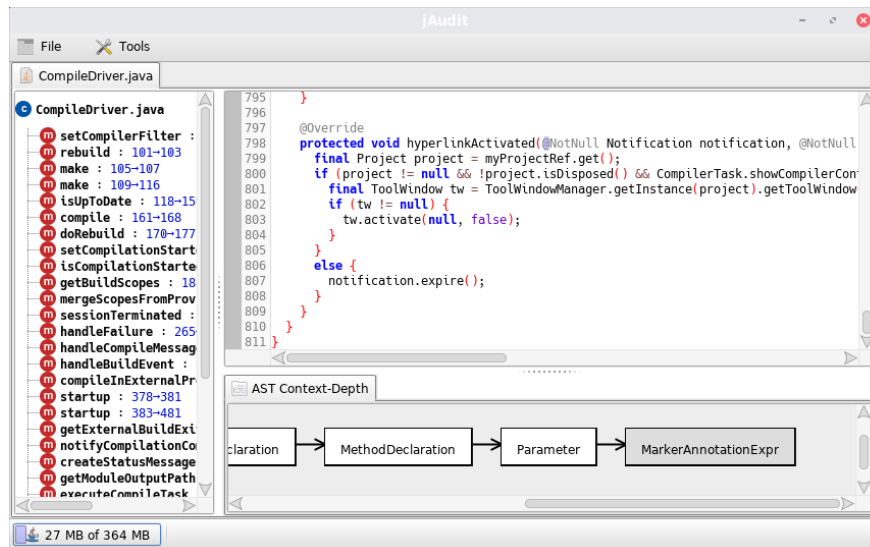


Figure 27: jAudit running under Linux

2.2 Profiling

In choosing Java as the development language of jAudit, I knew I'd have concerns over the JVM's notorious memory usage. In order to gain metrics regarding these worries, I decided to profile the JVM during the execution of a usual jAudit session.

For context, the JVM version used is shown below:

```
> $ java -version
openjdk version "1.8.0_144"
OpenJDK Runtime Environment (build 1.8.0_144-b01)
OpenJDK 64-Bit Server VM (build 25.144-b01, mixed mode)
```

2.2.1 Heap Overview

Ever since the inception of jAudit, I was adamant on the JVM memory usage component that you can see at the bottom of the main window's status bar. After seeing how memory usage seemed to fluctuate whilst performing different actions from within jAudit, I decided to see how the heap responded to the opening and parsing of files.

The graph below shows how the JVM heap memory fluctuated during the opening and parsing of 2 files in the one session:

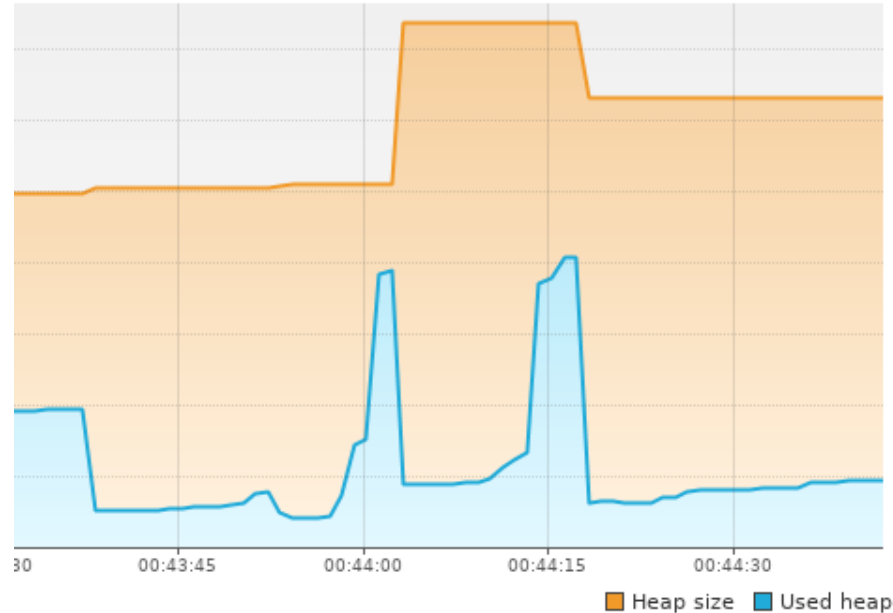


Figure 28: Heap fluctuation caused by parsing 2 files

As you can see, it's easy to work out when the two files were loaded. The two major spikes in the *Used Heap* show the sharp heap usage used by the parsing stage of jAudit. The first spike also noticeably shows how the *Heap Size* resizes to accommodate the spikes in heap usage. This extra allocated JVM heap memory acts as a safety net to avoid the JVM crashing with a `java.lang.OutOfMemoryError` exception.

I suspect that the JVM internally allocates this extra heap memory - the safety net - relative to the growth of used heap memory over time. This would explain why the JVM heap retains its rather excessive size long after the two spikes have occurred; e.g. if the JVM experiences a sharp increase in non-deterministic heap allocations, it will allocate an excessive heap size so it doesn't have to continually resize its own heap to account for volatile spikes in usage.

2.2.2 Largest Objects

Another metric that I believed would be quite interesting - given the amount of dependencies used in jAudit - were the largest objects allocated by each part of the program. I believed this metric would provide keen insight into the overhead caused by certain jAudit dependencies.

A part of jAudit that I suspected would have significant overhead instinctively - and partially after seeing the redraw lag on resizing the main window - is the look and feel provided by *WebLookAndFeel*.

WebLookAndFeel provides the look and feel used by jAudit - and a few of the components used. To work out the overhead imposed by this dependency, I used VisualVM to query the largest 20 objects related to the *MainView* context. The result of this query is shown below:

Class Name	Retained Size
com.alee.skin.web.WebSkin#1	3,217,603
com.alee.managers.style.data.SkinInfo#1	3,217,579
sun.misc.Launcher\$AppClassLoader#1	2,764,114
java.util.Vector#10	2,514,728
java.lang.Object[]#22066	2,514,692
class com.alee.managers.language.LanguageManager	975,597
class org.fife.ui.rsyntaxtextarea.modes.JavaTokenMaker	789,572
int[]#2454	635,736
com.alee.managers.language.data.Dictionary#1	419,488
java.util.ArrayList#8061	419,128
java.lang.Object[]#7669	419,096
class com.alee.utils.ReflectUtils	368,730
com.alee.extended.tree.AsyncTreeModel#1	335,018
com.alee.extended.tree.AsyncTreeModel#2	333,491
sun.awt.AppContext#1	310,830
java.util.HashMap#249	310,182
java.util.HashMap\$Node[]#4251	310,118
sun.awt.X11FontManager#1	266,565
com.alee.managers.language.data.Dictionary#2	264,432
java.util.ArrayList#8065	264,096

Figure 29: Largest objects used by the *MainView* instance

The objects I consider relevant, listed by order of size, are tabulated below:

Relevant MainView Overhead	
Instance Name	Size (bytes)
com.alee.skin.web.WebSkin	3,217,603
com.alee.managers.style.data.SkinInfo	3,217,579
com.alee.managers.language.LanguageManager	975,597
org.fife.ui.rsyntaxtextarea.modes.JavaTokenManager	789,572

From the table above, one can conclude that WebLookAndFeel is an expensive dependency to have on jAudit as it takes up more than 6 megabytes when idle (all

`com.altee.*` are WebLaF packages). Another, reasonably large, source of overhead comes from the `org.fife.ui.*` packages as they're responsible for providing the syntax-highlighting text component used for displaying code in jAudit.

2.2.3 Object Querying

Given the - admittedly nondescript - overview seen in the previous sections, I decided it would be interesting to query specific objects using the JVM's OQL (Object Query Language). I hoped the results of these queries would highlight where potential overhead is incurred.

Wasted String Overhead Since parsing source code is something that requires a lot of sectioned manipulation of strings, and the fact that Java's **Strings** are immutable in nature, I ran a predefined query to list strings whose memory overhead (`char[]`) is wasted due to operations such as substring, etc.

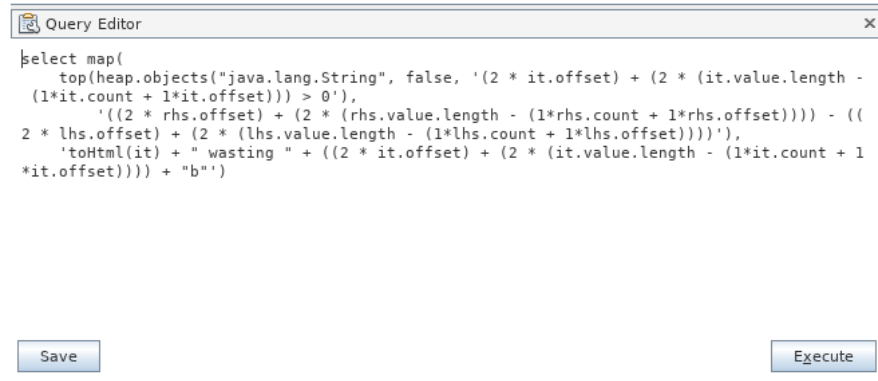


Figure 30: OQL query to find strings whose contents aren't fully used

The result of this query is shown below:

Query Results	
java.lang.String#32613	wasting 106960b
java.lang.String#32612	wasting 25380b
java.lang.String#25025	wasting 11328b
java.lang.String#32614	wasting 7540b
java.lang.String#25026	wasting 4096b
java.lang.String#24807	wasting 3720b
java.lang.String#611	wasting 3010b
java.lang.String#29433	wasting 2500b
java.lang.String#20133	wasting 2452b
java.lang.String#20129	wasting 2452b

Figure 31: OQL query to find strings whose contents aren't fully used

One can see from the results shown above that there isn't much wasted memory. However, I still attempted to see if it would be possible to reduce string wastage, so I decided to analyse the contents of the strings in an effort to work out what context within jAudit constructs or utilises them. Disappointingly, the contents of these strings were indecipherable spatterings of random ASCII and unicode characters so no context could be inferred from them.

File Instance Overhead Another query that I believed would be interesting would be a query that queries for all *File* objects as I suspected operations such as opening files - which require an open file dialog - would incur temporary memory overhead.

I constructed a rather simple query designed purely to select all instances of the `java.io.File` class:

```
select x from java.io.File x
```

The results of this query are partially shown below:

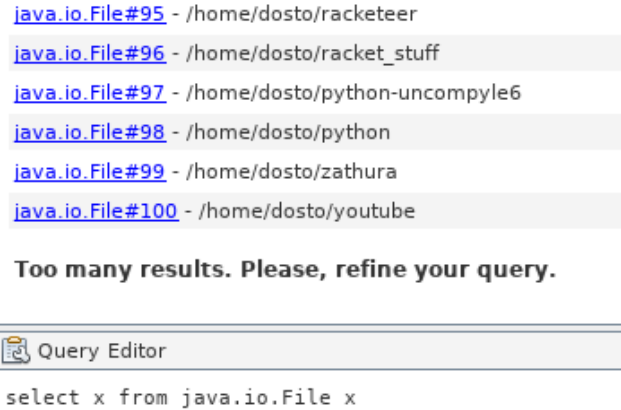


Figure 32: Partial results of File query

As you can see, the results are truncated due to there being too many results.

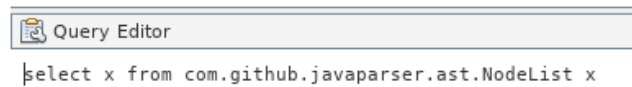
It didn't take long to work out the creational context of all of these `File` instances since they were unrelated to the direct actions used in `jAudit`. Due to this fact, it's evident that these instances are used by the open file dialog used to open files in `jAudit`. One could argue that an instance of a class like `File` is simply extra baggage for the open file dialog to allocate so many of. However, when you actually consider the information that must be known to the open file dialog, it's hard to think of an alternative class one could create that doesn't end up incurring near-enough the same overhead anyway. Thus, I decided the overhead wasn't that significant and that the JVM's garbage collector should hopefully clear those unreferenced instances after the open file action is complete.

2.2.4 Querying NodeList

Having queried various objects in my profiling efforts, I finally chose to see the overhead incurred by the hierarchical, cyclic-referencing, structure of AST nodes as the AST forms a major part of `jAudit`'s internal functionality.

I adapted the previous query used for querying `java.io.File` instances to search for `com.github.javaparser.ast.NodeList` instances instead as this class forms the major hierarchical component used for storing AST nodes.

```
select x from java.io.File x
```

Figure 33: Query for finding `NodeList` instances

The query yielded similar large results like the other queries so I decided to take a look at the overview of the `NodeList` class in memory. The overview of one of these `NodeLists` is shown below:

com.github.javaparser.ast.NodeList Instances: 1,806 | Instance si

Instances		Fields
Instance	Retained	Field
#1	104	this
#2	136	observers
#3	104	parentNode
#4	136	innerList
#5	104	size
#6	104	elementData
#7	136	[0]
#8	136	modCount

Field Type

- this: NodeList
- observers: ArrayList
- parentNode: ClassOrInterfaceType
- innerList: ArrayList
- size: int
- elementData: Object[]
- [0]: ClassOrInterfaceType
- modCount: int

Figure 34: `NodeList` overview

One can tell from the figure above that `NodeList` and associated classes contain cyclic references as part of their ownership hierarchy. This is often considered bad practice in other languages like C++ where the child would often just reference a weak pointer that doesn't own the object it's referencing. However, in Java, we're stuck with no real ownership model due to lack of disparity in class instances.

An example referential hierarchy (starting from `MainView` - the primary view in `jAudit`) can be seen below:

Field	Type	Value
this	SimpleName	#1508
name	VariableDeclarator	#117
item	LinkedList\$Node	#4467
last	LinkedList	#8370
childNodes	VariableDeclaration...	#109
variable	ForeachStmt	#9
[3]	Object[]	#23552 9 items
elementData	ArrayList	#25401
innerList	NodeList	#1473
statements	BlockStmt	#145
body	MethodDeclaration	#31
node	MethodTreeNode	#31
[29]	Object[]	#23525 40 items
elementData	Vector	#644
children	ClassTreeNode	#1
root	DefaultTreeModel	#1
treeModel	AuditModel	#1
model	AuditView	#1
visComp	JTabbedPane	#1
tabPane	MainView	#1 jAudit

Figure 35: NodeList overview

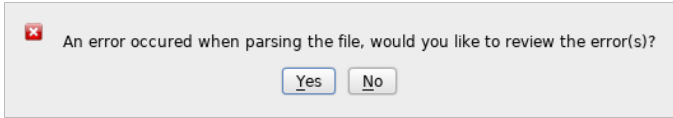
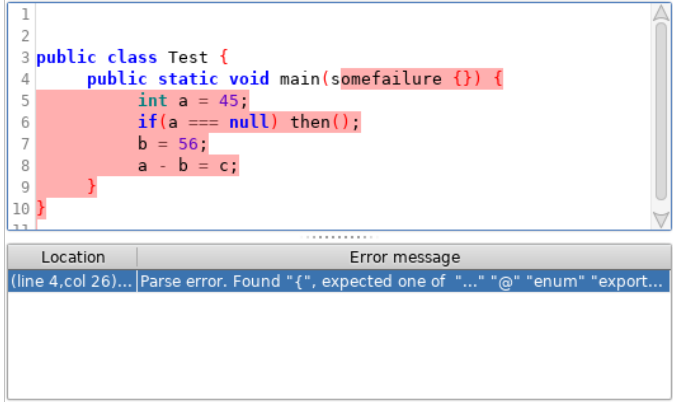
Having some understanding of how the JVM's garbage collector works, I've concluded that there is definitely memory overhead as a result of this hierarchy. The JVM considers objects as *garbage* if they aren't reachable from traversing an initial collection root. I believe the fact that the majority of the AST is continually referenced in memory (be it directly or indirectly) means the garbage collector can't do much for us when it comes to optimising memory usage. However, one could conceivably explicitly nullify parents to break pre-existing ownership hierarchies and implement sufficient error-prevention in order for the JVM to consider unused elements of the AST as garbage. A simpler way to decrease memory consumption would be to implement partial extraction of AST elements within a context that only parses certain elements of the source code at a time (when needed - this incurs a performance penalty though). However, due to implementation constraints within JavaParser - e.g. the library jAudit uses for parsing, this isn't a feasible approach.

I found that the average size of a NodeList instance was 118 bytes.

2.3 Functional testing

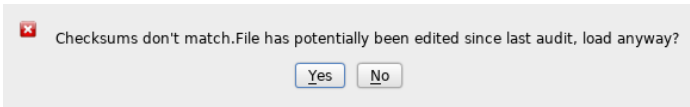
2.3.1 Testing of Invalid Source Files

In order to test the functionality of the parse problem view, I had to manually supply an invalid source file. This was a form of functional testing to prove the functional logic of jAudit when it encounters a file it fails to parse.

Functional testing of ParseProblemView	
Scope: the response of jAudit when given an invalid source file to parse.	
Input: invalid Java source file	
<pre> 1 public class Test { 2 public static void main(somefailure {}) { 3 int a = 45; 4 if(a === null) then(); 5 b = 56; 6 a - b = c; 7 } 8 } </pre>	
Expected behaviour: user should be notified of the parser's inability to parse the file and prompted with the option to view the parser's reported errors.	
Actual behaviour: user is shown an error message that asks them if they'd like to review the errors the parser encountered	
	
Figure 36: Error message shown when file can't be parsed	
	
Figure 37: Error prompt shown to notify user of contained error	
Conclusion: user is correctly shown an error message and prompted for a review - this functionality matches the functional specification.	

2.3.2 Testing of Differing Checksums

I also thought it was important to test the functional logic of how jAudit handles cases where files already stored in the database are reloaded with their checksums differing.

Functional testing of different checksums for same file	
Scope:	to evaluate how jAudit responds to receiving an already-referenced file whose checksum doesn't match the one previously calculated for it.
Input:	modified file already referenced in database; e.g. file with differing checksum from one already stored for it
Expected behaviour:	user is warned of checksum mismatch and retains the ability to still continue with the opening of the file
Actual behaviour:	user is notified of checksum mismatch and can still continue to open the file
	
Figure 38: Checksum warning	
Conclusion:	user is correctly notified of checksum mismatch but retains the ability to continue in opening the file - this behaviour is expected and matches the specification as it relates to allowing the target demographic to retain control and continue in spite of warnings

2.4 Migrations testing

As mentioned before, jAudit's database construction relies on a migrations pattern where each migration consists of the database operation to construct a table and the relevant operation to destroy it.

I didn't believe there was any value in any effort to formally verify the correctness of each migration as the verification system wouldn't be external. Thus, any doubts regarding the validity of the migrations could be shared regarding the validity of the verification system.

Technically, a unit test could've been drawn up because the migrations catch exceptions and do nothing with them; so some type of fail flag could've been set. However, I chose not to write unit tests because the most likely reason as to why a migration would fail would be a reason related to database connectivity, not the logic of the migration. So a unit test wouldn't really be proving anything if the injected connection is invalid in the first place.

Hence I chose to manually verify the output:

```
1 // create file table
2 success = connection.query("CREATE TABLE IF NOT EXISTS files (\n" +
3     "file_id integer PRIMARY KEY,\n" +
4     "path text NOT NULL,\n" +
5     "checksum integer NOT NULL,\n" +
6     "date_added integer NOT NULL)"
7 );
8
9 // create audit table
10 success = connection.query("CREATE TABLE IF NOT EXISTS audits (\n" +
11     "audit_id integer PRIMARY KEY,\n" +
12     "file_id integer NOT NULL,\n" +
13     "begin integer NOT NULL,\n" +
14     "end integer NOT NULL,\n" +
15     "context text NOT NULL,\n" +
16     "comment text NOT NULL)"
17 );
```

After running both of those migrations on a database, named *jaudit.sqlite*, I manually dumped the schema and did a direct comparison:

```
> $ sqlite3 jaudit.sqlite
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE files (
  file_id integer PRIMARY KEY,
  path text NOT NULL,
  checksum integer NOT NULL,
  date_added integer NOT NULL);
CREATE TABLE audits (
  audit_id integer PRIMARY KEY,
  file_id integer NOT NULL,
  begin integer NOT NULL,
  end integer NOT NULL,
  context text NOT NULL,
  comment text NOT NULL);
sqlite>
```

As you can see, the dumped schema matches the migrations' queries. Again, I

didn't see the need to formally verify the migrations for aforementioned reasons and the fact that jAudit's database system is rather minimal; there's no ORM (Object Relational Mapping). Such a layer of abstraction could've potentially led to formal verification being required. At this stage, however, I deem manual verification of the testing of migrations to be sufficient.

2.5 Test plan

2.5.1 Unit Test: GitHub Verifier

I decided it would be best to formally unit test the GitHub verifier class to confirm the correctness of what's deemed as valid input.

The unit test consists of 2 sets, a set of valid input forms, and a set of invalid input forms. The unit test simply creates a verifier and iterates each set, asserting that each one respectively yields the expected result based on an assertion made upon the return value of the verifier.

Relevant MainView Overhead
Valid set
http://github.com/user/repo/file.java https://github.com/user/repo/file.java https://github.com/user/repo/file.java https://github.com/user/repo/file.java https://github.com/user/repo/file.java http://github.com/user/repo/file.java
Invalid set
', ' aaa google.com http://github.com/ https://github.com/ https://github.com/user https://github.com/user/ https://github.com/user// https://github.com/user/repo/ https://github.com/user/repo/file https://github.com/user/repo/.java https://github.com/user/repo/File.java_dangling

The code that iterates and asserts their respective validity is rather simple:

```

1  /**
2   * Test that expected cases pass
3   */
4   @Test
5   public void verifyInput() {
6       // assert that all expected-to-fail cases fail
7       for(String input : invalidSet)
8           assertFalse("Assertion that input is invalid!",
9                       verifier.verifyInput(input));
10
11      // assert that all expected to pass cases pass
12      for(String input : validSet)
13          assertTrue("Assertion that input is valid",
14                    verifier.verifyInput(input));
15  }

```

As this method is annotated as a test case with `@Test`, it can be targeted from within my IDE's `jUnit` integration feature:

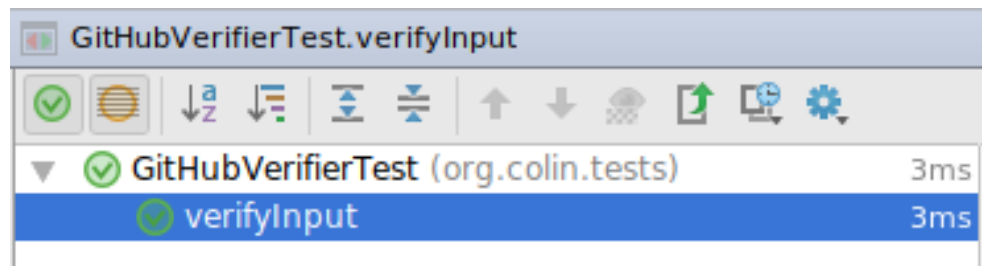


Figure 39: Targeted test case from IDEA

After running this test:

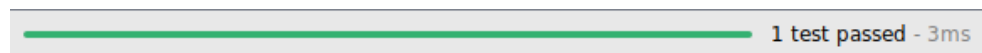


Figure 40: Targeted test case from IDEA

As you can see, the test passed within a few milliseconds. The fact that it only takes a few milliseconds could also be used to verify the suitability of the unit being tested as a realtime input validation component.

2.5.2 Unit Test: Audit Model Test

The audit model stores a copy of the file ID it's working with. I decided to add an argument constraint where you can't set an invalid file ID. An *invalid* file ID would

be a value that can't be used as a primary key ID in a database; e.g. any value less than 0 is invalid.

I implemented this constraint using an `IllegalArgumentException` that would be thrown if the parameter `fileId` is less than 0.

Scope: the scope of this unit test is the `setFileId` method as it logically shouldn't accept an input that can't be a valid - integral - primary key in a database. By ensuring the validity of the model's file ID, we can form a first line of defence against invalid file referencing; a model referring to a file which is not yet stored in a database has an ID of -1.

```
1  /**
2   * Set file ID
3   * @param fileId database row ID
4   */
5  public void setFileId(int fileId) throws IllegalArgumentException {
6      if(fileId < 0)
7          throw new IllegalArgumentException("ID must be >= 0");
8
9      this.fileId = fileId;
10 }
```

```
1  /**
2   * Attempt to set an invalid file ID.
3   * The very first invalid case being (-1).
4   * This test case expects an {@link IllegalArgumentException} to be thrown.
5   */
6  @Test(expected = java.lang.IllegalArgumentException.class)
7  public void setFileId() {
8      model.setFileId(-1);
9  }
```

The testing of this specific component is documented below:

Test cases		
Input	Expected	Actual
Normal		
123	no exception thrown	no exception thrown
1337	no exception thrown	no exception thrown
1	no exception thrown	no exception thrown
Extreme		
0	no exception thrown	no exception thrown
2,147,483,647	no exception thrown	no exception thrown
Exceptional		
-1	exception thrown	IllegalArgumentException
-2,147,483,648	exception thrown	IllegalArgumentException
"123"	code doesn't compile	code doesn't compile

Conclusion: I conclude from unit testing - documented above - that the component performs as designed - all normal input succeeds, as do the extreme cases. Exceptional cases correctly throw exceptions or can't be compiled at all.

2.5.3 Unit Test: MethodTreeVisitor

I opted to test the validity of the method-tree visitor by testing that it correctly visits every method in a compilation unit. To test this, I simply wrote a test that dynamically generates a compilation unit with methods whose names are listed in an input array. Then, after visitation, the test would assert that the methods it visited match the ones created dynamically.

Scope: the scope of the method tree visitor is to ascertain the validity of the visitor - we assume there is no semantic difference between a dynamically constructed compilation unit and one constructed from a file. The method-tree visitor is used to populate the method-tree used in the audit view.

```

1 package org.colin.visitors;
2
3 import com.github.javaparser.ast.CompilationUnit;
4 import com.github.javaparser.ast.Modifier;
5 import com.github.javaparser.ast.body.ClassOrInterfaceDeclaration;
6 import com.github.javaparser.ast.body.MethodDeclaration;
7 import org.colin.gui.ClassTreeNode;
8 import org.colin.gui.MethodTreeNode;
9 import org.junit.Before;
10 import org.junit.Test;
11
12 import static org.junit.Assert.*;
13

```

```

14  /**
15   * Unit test of {@link MethodTreeVisitor} to verify that it visits all methods in
16   * a compilation unit correctly. This class extends the class it's visiting to
17   * simply print out diagnostic information when the test is ran.
18   */
19  public class MethodTreeVisitorTest extends MethodTreeVisitor {
20      /**
21       * Compilation unit containing declarations
22       */
23      private CompilationUnit unit;
24
25      /**
26       * Dummy name for class we're creating dynamically within unit.
27       */
28      private final String CLASS_NAME = "Class";
29
30      /**
31       * Method test data (names of methods to dynamically create)
32       */
33      private String[] methods = { "method", "doSomething", "moreMethods" };
34
35      /**
36       * Initialise compilation unit and dynamically populate class declaration
37       * with methods listed in {@link MethodTreeVisitorTest#methods}.
38       */
39      @Before
40      public void setUp() {
41          // initialise compilation unit
42          unit = new CompilationUnit();
43
44          // create a dummy class declaration
45          final ClassOrInterfaceDeclaration classDecl = unit.addClass(CLASS_NAME);
46
47          // iterate and dynamically append method declarations to dummy
48          // class declaration
49          for(final String methodName : methods)
50              classDecl.addMethod(methodName, Modifier.PUBLIC);
51      }
52
53      /**
54       * Visit the compilation and assert that the extracted methods from the
55       * visitor match the ones we've dynamically inserted during
56       * {@link MethodTreeVisitorTest#setUp()}.
57       */

```

```

58     @Test
59     public void visit() {
60         // create root class node for visitor to append to
61         ClassTreeNode classNode = new ClassTreeNode(CLASS_NAME);
62
63         // call our (essentially policy-free) diagnostic wrapper
64         this.visit(unit, classNode);
65
66         // iterate root class' mtehod(s) and assert that they match the
67         // names of those we created
68         for(int i = 0; i < classNode.getChildCount(); i++) {
69             // get method tree node
70             MethodTreeNode visitedNode = (MethodTreeNode) classNode.getChildAt(i);
71
72             // assert it matches what we created
73             assertTrue(methods[i].equals(visitedNode.getName()));
74         }
75     }
76
77     /**
78     * Print diagnostic information (of what's being parsed)
79     * @param methodDecl method declaration node
80     * @param root tree node being appended to (where the methods are leaves)
81     */
82     @Override
83     public void visit(MethodDeclaration methodDecl, ClassTreeNode root) {
84         // visit implementation of class we're testing
85         super.visit(methodDecl, root);
86
87         // print out diagnostics
88         System.out.println("Visiting: " + methodDecl.getName());
89     }
90 }

```

The dynamically generated compilation unit created in the test above outputs:

```

1  public class Class {
2
3      public void method() {
4      }
5

```



```

6   public void doSomething() {
7   }
8
9   public void moreMethods() {
10  }
11 }

```

This output is then fed to the visitor (in the form of a `NodeList` hierarchy) which, when ran in IntelliJ's testing facility, outputs:

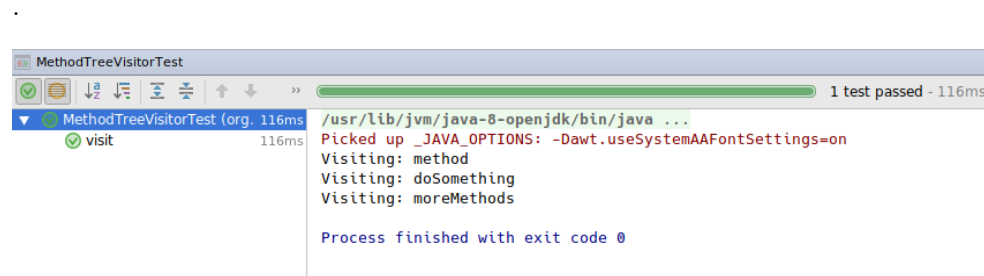


Figure 41: MethodTreeVisitor output

Conclusion: As shown above, the visitor correctly visits the dynamically created methods. This, in my view, proves the validity of the unit - visitor - being tested.

3 Documentation

3.1 Online help features

3.1.1 FAQ Website

In order to clarify certain questions I believe potential users would have before or after using jAudit, I decided to create online help features in the form of a FAQ (Frequently Asked Questions) website.

A screenshot of this FAQ site is shown below:

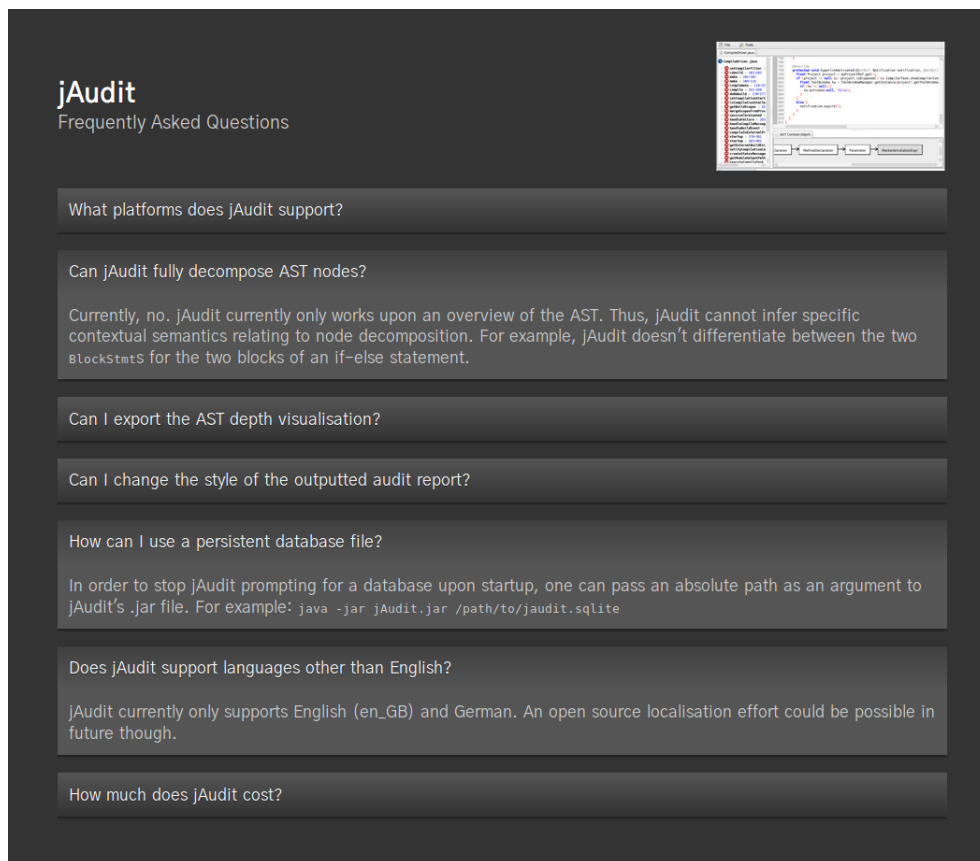


Figure 42: FAQ Website

Each question's associated answer can be revealed by simply clicking the question - this causes the answer panel to slide down and become visible.

3.2 This Document

This document can also act as a relevant help guide (or other form of documentation) as I believe it outlines motivations, implementation details, etc. that could be potentially useful for both users and any developers who wish to contribute to jAudit.