Team D - Milestone 4- Final Document

Isaac Friedman, Lucas Pickens, Kati Jurgens, and Michael Webb

May 13th, 2022

Introduction:

This document below describes the workings of the Team D Java Linter project. This project's goal is to provide a tool that takes in Java code and detects various style/design violations and design patterns, providing useful output to the user. This tool is mainly meant for use by developers familiar with the Eclipse environment, design patterns, and in some cases, Github. The software should work on any hardware, but some of the functionality we have implemented means it can only run in the Eclipse application. This means any developer wishing to use our program must have access to a system that can run Eclipse, but since it is a free IDE, cost should be no object. This document is organized into sections, as referenced in the table of contents, split between user and system developer use. The user documents, specifically the Getting Started Guide, are a good way for a user to familiarize themselves with the system, with help from the Installation/Configuration parts of the Maintenance Guide. For a deeper look on how the system works internally, a developer may reference the Maintenance Guide, the Software Requirements Specification (SRS), the Software and Architecture Design Specification (SADS), and the Test Plan. Overall, the point of this document is to better acquaint any form of user interacting with the system with whatever part they wish to know about.

User Documents:

User Guide

1. System Overview

The Java Linter is a downloadable application run through an IDE that provides the following nine lint checks to be run on Java code:

- Poor naming of classes, variables and methods
- Long methods
- Variables instantiated but not used
- Unnecessary Interfaces
- Violations of the Hollywood Principle
- Code that favors inheritance over composition
- Detection of Strategy Pattern
- Detection of Facade Pattern
- Detection of Adapter Pattern

The application can be run through the IDE console or can be used as a GUI application. Code to be checked can be added to the linter by adding a package to the linter project or through a Github link.

2. User Tasks

2.1 Starting the System

To start the system, run the InputManager file in the Presentation Package

2.2 Running the System From the Console

Once the system has started, when prompted to choose an access method, enter "Console" to continue using the system through the IDE console

2.2.1 Adding Local Packages to Lint

When prompted to choose the import method, enter "Package" to add a local package, then enter the name of the desired package when prompted. When prompted for whether you would like to enter another package, enter either "y" or "n" and then enter the name of the desired package when prompted.

2.2.2 Adding files from Github to Lint

When prompted to choose the input method, enter "Github" to add files from Github, then enter a link to the desired class or package when prompted. The link should be

formatted so the end of the url points to either a .java class or a package. When prompted for whether you would like to enter another github link, enter either “y” or “n” and then enter the name of the desired class or package when prompted.

2.2.3 Adding Lint Checks to Run

When prompted to import a check to add, enter any of the following to add the corresponding lint check

- name - Bad Names Check
- instantiation - Unused Instantiation Check
- length - Method Length Check
- hollywood - Hollywood Principle Violations Check
- composition - Composition Over Inheritance Check
- interface - Redundant Interface Check
- strategy - Strategy Pattern Detector
- facade - Facade Pattern Detector
- adapter - Adapter Pattern Detector
- All - Adds all above checks

2.2.4 Removing Classes Staged to Lint

When prompted to import a check to add, enter “remove”. You will be shown a list of all classes currently staged to lint. Enter the name of one to remove it from the staged files

2.2.5 Re-add Class Removed From Lint Stage

When prompted to import a check to add, enter “readd”. You will be shown a list of classes that have been staged and removed. Enter the name of one to stage it to be linted

2.2.6 View Help Menu

When prompted to import a check to add, enter “help”. You will be shown a list of all commands as seen below:

```
Input the name of the check you would like to run:help
Possible commands:
'all' : Runs all of the checks listed below
'name' : Checks for name convention violations in classes, methods, and fields
'instantiation' : Checks for variables and fields that are instantiated but never used
'length' : Checks for methods that are too long
'hollywood' : Checks for violations of the Hollywood Principle
'composition' : Checks for places where composition could be used instead of inheritance
'interface' : Checks for interfaces that might be redundant
'strategy' : Detects use of Strategy pattern
'adapter' :  Detects use of Adapter pattern
'facade' : Detects use of Facade pattern
'run' : Runs the checks that have been added
'remove' : Select classes added to list of classes to be tested to be removed from list
'readd' : Select classes previously removed from list of classes to be tested to be added back to list
'exit'/'done' : Exits the program
Input the name of the check you would like to run:
```

2.2.7 Run Selected Checks on Staged Classes

When prompted to import a check to add, enter “run”. The system will run the selected checks on the staged files and display the output as well as write linter output to the file “LintOutput.txt”

2.2.7 Exit Program

When prompted to import a check to add, enter “exit” or “done”. The program will end.

2.3 Running the System Through the GUI

Once the system has started, when prompted to choose an access method, enter “GUI” to access the graphical interface

2.3.1 Adding Local Packages To Lint

When prompted to choose an import method, press the “Packages” button. When prompted for a package name, enter the name of the package you wish to stage to be linted and press the “Add” Button. Repeat this for each package you wish to add, then press the “Submit” button.

2.3.2 Adding Files From Github to Lint

When prompted to choose an import method, press the “Github” button. When prompted for a link, enter the github link to the desired class or package and press the “Add” button. The link should be formatted so the end of the url points to either a .java class or a package. Repeat this for each class or package you wish to add, then press the “Submit” button

2.3.3 Remove Classes Staged to be Linted

When the system displays a list of classes staged to be linted, enter the name of the class you wish to remove from staging and press the “Remove” button

2.3.4 Re-add Classes Removed from Lint Stage

When the system displays a list of classes staged to be linted, enter the name of the class you wish to add to staging and press the “Add” button

2.3.5 Proceed to Lint Check Selection

When the system displays a list of classes staged to be linted, press the “Submit” button to confirm classes to be linted.

2.3.6 Select Lint Checks

When the system displays a list of lint checks, click the lint check you wish to run, then press the “Submit” button. If you wish to add multiple checks, hold the Ctrl key and click each check to be run.

2.3.7 Run Lint Checks

When the system displays a list of lint checks, press the “Submit” button to run the selected lint checks on the selected classes. You will be shown the linter output.

2.3.8 Restart Linter

When the system displays the linter output, press the “Restart” button. The program will return to import method selection

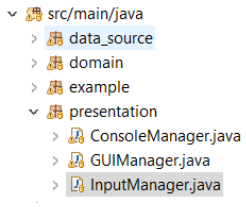
2.3.9 Exit Program

When the system displays the linter output, press the “Exit” button. The program will exit.

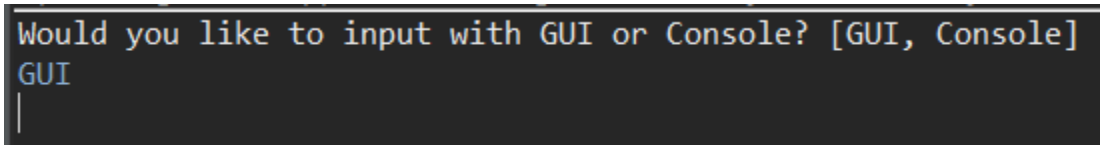
Getting Started Guide

For first time users, the GUI provides a more easily understood view of the application.

Starting the Application:

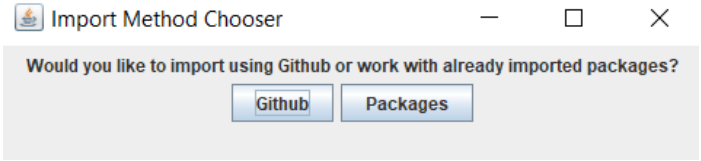


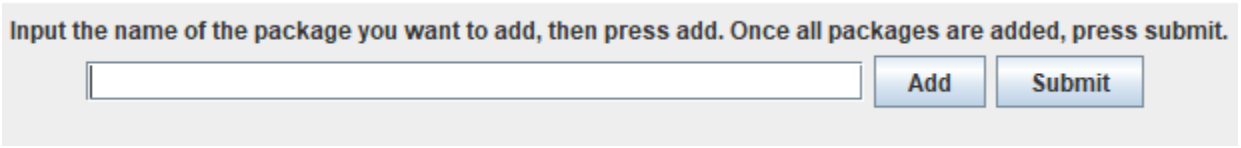
To start the application, run the file “InputManager” located within the package “src/main/java/presentation” as a Java Application. After starting the application, the console will prompt the user to continue in the console or a GUI. Enter “GUI”.



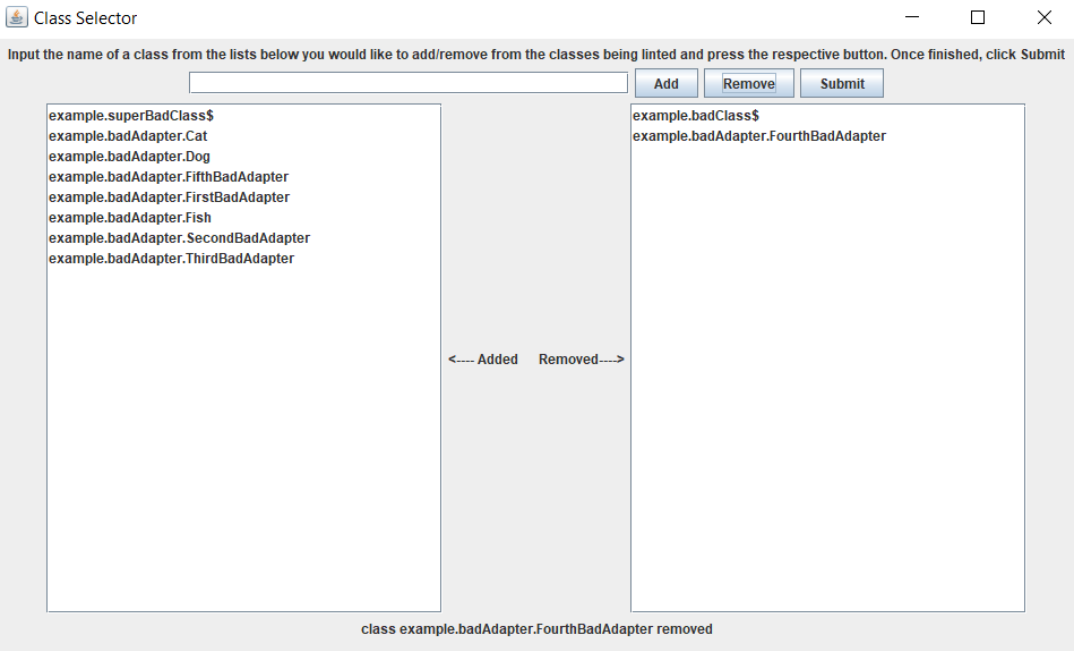
Importing Classes to Lint:

When Importing classes to lint into the application, two options are given. Users can either import classes that are already locally in the project by package, or provide the application with files via a link to a github class or package.

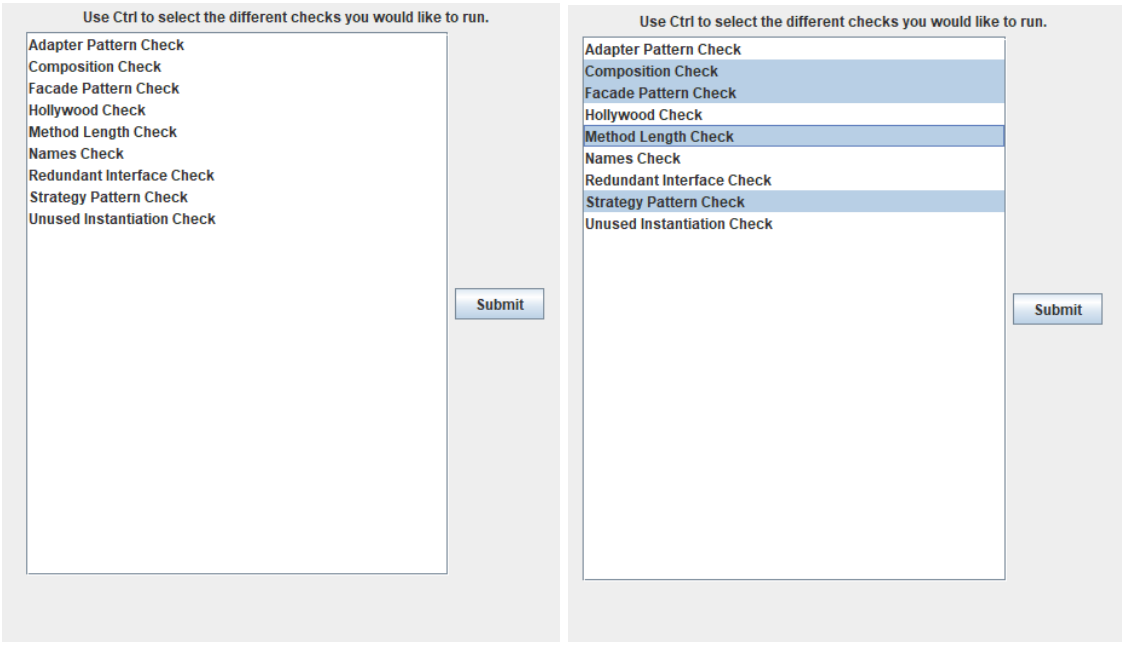




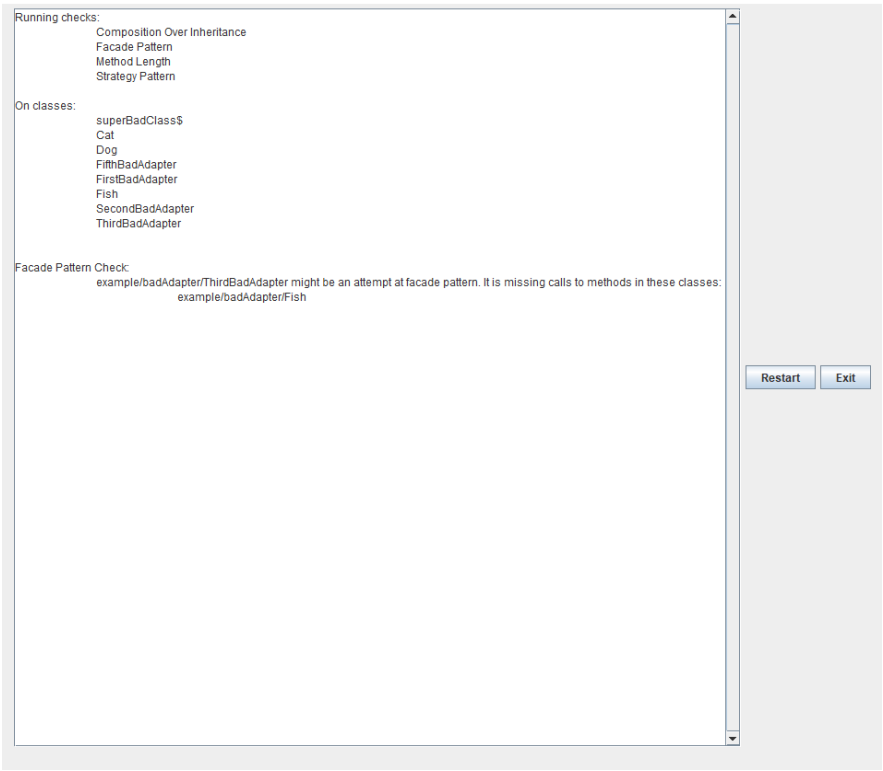
Using either import method, the user will enter as many packages or links as desired, then press submit to continue. The application will then display the Class Selector screen where the user can remove and re-add classes to be linted from the classes and packages provided before submitting the class selection



After confirming which classes are to be linted, the application will prompt the user to choose which checks to run. Hold the control key and click each check to be run, then click submit



The application will run the selected checks on the given classes and display the output along with options to exit the application or restart from selecting an import method.


Running checks:
Composition Over Inheritance
Facade Pattern
Method Length
Strategy Pattern
On classes:
superBadClass$
Cat
Dog
FifthBadAdapter
FirstBadAdapter
Fish
SecondBadAdapter
ThirdBadAdapter
Facade Pattern Check:
example/badAdapter/ThirdBadAdapter might be an attempt at facade pattern. It is missing calls to methods in these classes:
example/badAdapter/Fish
Restart
Exit

Installation, Configuration, and Maintenance Guide:

Purpose

The purpose of this installation, configuration, and maintenance guide is for the future developers of this Java Linter to be able to understand previously written code and know where to start to make changes. It will have a brief section on how to install the code and how to configure the environment to run this program, as well as steps to take for maintaining the source code.

Overview

This Java Linter is a program that allows users to input Java packages and classes from either the local system, or by downloading from a Github link to be checked for certain design and syntax features. The linter has the ability to check for badly named objects, too long of methods, unused instantiation of variables, redundant interfaces, Hollywood principle violations, using inheritance instead of composition, Strategy Pattern, Adapter Pattern, and Facade Pattern.

More details about how to use this software can be found in the user documents

Installation

This is a simple guide to get our application working on your machine.

1. Download and unzip the download package
2. Open your preferred Eclipse Workspace
3. Create a Gradle Project and name it whatever you wish
4. Delete any pre-generated main or test files
5. Import the CSSE375.jar file as an Archive File into the project
6. Before finishing the import, change your “Into Folder” path to:
   Project Name/src/main/java
7. Drag in the new build.gradle file
8. Open the project’s build path
9. Click “Add External JARs” under the “Libraries” tab
10. Import the dependencies from the dependencies folder

You now have a fully running project!

Configuration

In terms of configuration, the only task required is to enable native polling. This is done through Eclipse, by navigating the menus as described below:

Window > **Preferences** > **General** > **Workspace**

Enable “Refresh Using native hooks or polling.”

Libraries and Technologies

This project does not use many libraries or outside technologies but the ones that are used can be pretty dense to work with. Each library/technology has a high-level training strategy for tips on how to get started with the technology.

ASM

The basis of this project is built on the open-source software ASM developed by OW2. This software allows us to analyze Java classes and the different features within each class. ASM is a large third-party program that holds more information than we needed for the scope of our project so we wrapped parts of ASM in our own version of ASM. To get an understanding of how the underlying software works, look at the suggest materials below:

Training Strategy

- Read the website for ASM https://asm.ow2.io/index.html and focus on the following:
    - User Guide: https://asm.ow2.io/asm4-guide.pdf (Sections 2 and 3)
    - Frequently Asked Questions: https://asm.ow2.io/faq.html
- Download the most recent release of the ASM jar files (“asm-##.jar”, “asm-tree-##.jar”, “asm-analysis-##.jar”) from:
    - https://repository.ow2.org/nexus/content/repositories/releases/org/ow2/asm/asm/
    - https://repository.ow2.org/nexus/content/repositories/releases/org/ow2/asm/asm-tree/
    - https://repository.ow2.org/nexus/content/repositories/releases/org/ow2/asm/asm-analysis/
- Create a test program and try out different features of ASM on an existing Java file
- The Eclipse plugin Bytecode Outline is also a good feature to help learn how ASM works https://marketplace.eclipse.org/content/bytecode-outline

Gradle

This project uses the build automation and dependency manager tool called Gradle. This allows for easier organization of files and test cases within the file system. To get started with this software, below are suggested materials:

Training Strategy

- Read the Gradle User Manual: https://docs.gradle.org/current/userguide/userguide.html
- Download Gradle at https://gradle.org/install/
- Ensure that it is set up within Eclipse properly by following this tutorial: https://www.vogella.com/tutorials/EclipseGradle/article.html

Github

This project uses Github as its version control software for tracking changes over time. Currently there is a private repository, but in the future, this could be switched to a different repository depending on what changes are made. For those who have never used Github before, below are suggested places to start:

Training Strategy

- Getting Started Documentation: https://docs.github.com/en/get-started
- Basic Commands: https://docs.github.com/en/get-started/using-git/about-git
- Create your own small Hello World project and test out some of the commands listed above

JSON

This project uses a small JSON library to import Github links and convert them to a readable JSON format. This enables us to be able to read any Java project or folder on Github. In the future we could enable functionality to read entire Java projects stored on Github or any other website that offers a JSON output.

Training Strategy

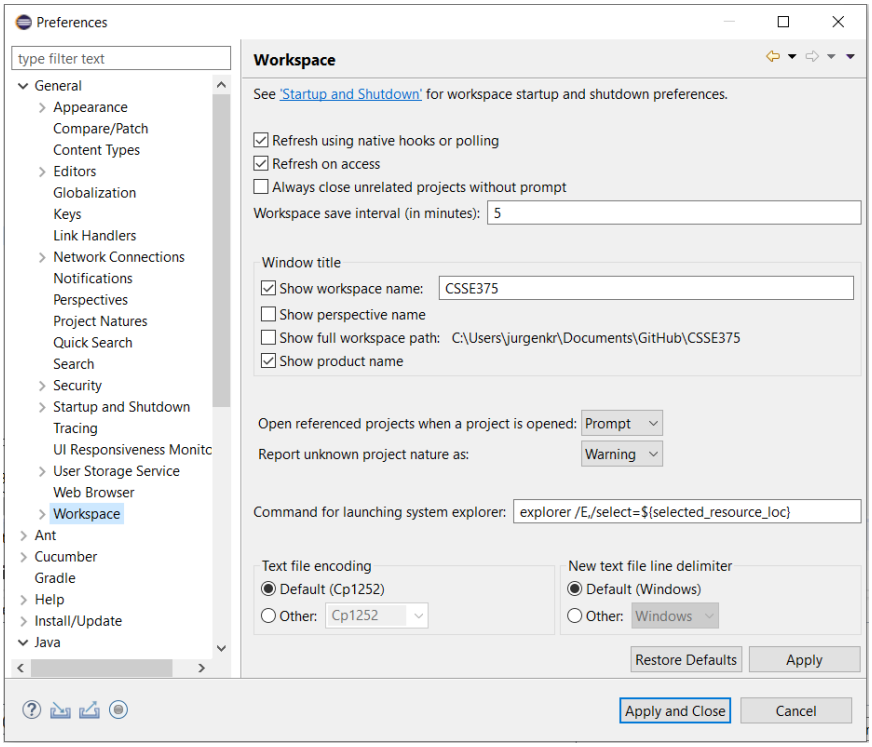
- Reference Documentation: https://docs.oracle.com/javaee/7/api/index.html
- Useful Quickstart Guide: https://www.baeldung.com/java-org-json

Development Environment

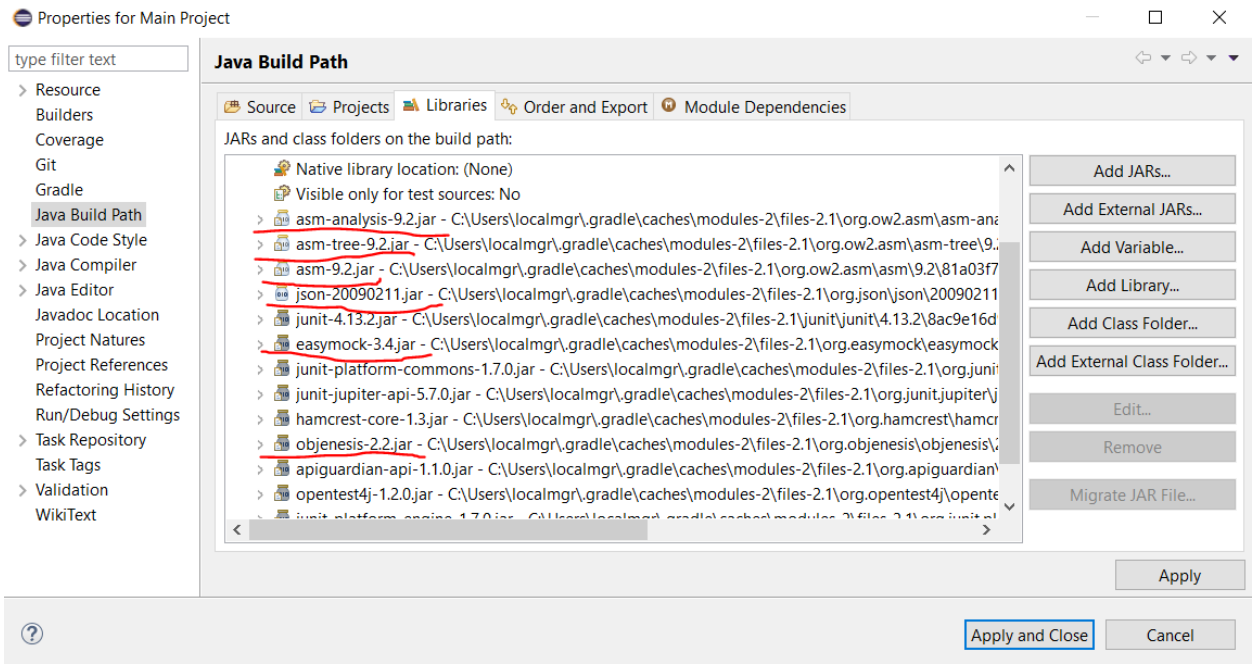
This project was developed using the environment Eclipse. We have the version Eclipse Java 2019-2020, however any version that is more recent should be compatible. For all

features of the project to work, it will be necessary to change the following settings in Eclipse:

- Window -> Preferences -> General -> Workspace
    - Ensure that “Refresh using native hooks or polling” and “Refresh on access” are selected



- CSSE375-LinterProject -> Properties -> Java Build Path
    - Ensure that a version of ASM, ASM-Analysis, ASM-Tree, JSON, EasyMock, and Objenesis is listed in the Project and External Dependencies



How To:

Below are basic high-level walkthroughs for how to do common tasks within the software.

Add New Check Type

Each check is its own class that is inheriting from the ClassCheck interface. To create a new check:

- Create a new class that inherits from the ClassCheck interface
    - Use the augmented ASM classes (My**) to analyze the code for the check
    - Ensure that the return value follows the format of if there are no violations or errors, it returns an empty string
- Add the new check as an option in the GUIManager and ConsoleManager
- Update the help menu information to include the new check

Update MyASM Interpretation

MyASM is the collection of classes (My**) that decorate or augment the original ASM classes. This is to ensure that any changes that ASM makes won't completely break our system and if we decide to switch to a different underlying system than ASM, it is easier to change the code. While continuing development, if it is noted that a change should be made to the MyASM classes, follow the steps below:

- Notate where this will affect the code and test code
    - If this is a naming change, simple replace text works
    - If the logic of the code will work, ensure that all tests still pass after changes are made

- Ensure that all copies of the old code are changed to the new code
- Ensure that test cases that include EasyMock are also updated with the new information

Possible Future Work

- Add more styles of linter checks
- Update the GUI

Troubleshooting Guide

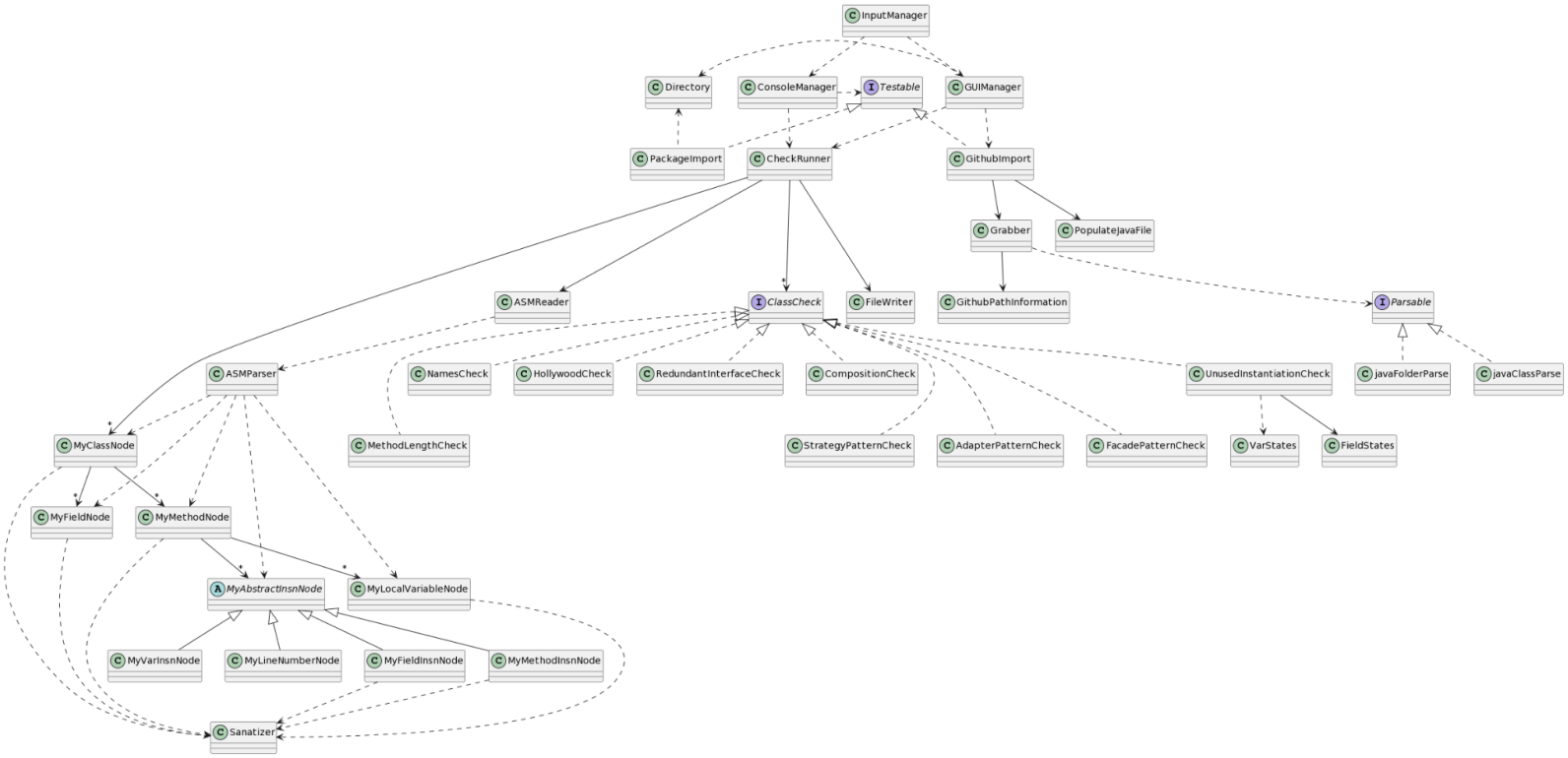
Issue	**Possible Causes**	**Suggested Action**
The system fails with an exception	The Github link that the system is trying to import is invalid	Try a valid link to a file/package.
	The code trying to be linted does not compile (i.e. File name is different than class name)	Fix the code so it does compile
	The github files are taking longer than expected to compile in Eclipse	Wait for longer before trying to advance past importing files.
Importing the project through gradle does not follow the specifications exactly.	Your gradle may create a new project as two separate projects in the workspace	Work in the one titled “lib,” as it is technically a nested project for the project you created
The program is still running when you do not expect it to.	The GUI program was closed partway through the linting process, instead of at the end.	Terminate the program manually through Eclipse
Files created by importing from Github were not deleted	A run did not completely finish or the run failed at some point.	Manually delete the imported files from the system

Software Requirements Specification (SRS):

Requirement	Description
R0	Provides 9 different lint checks to be run on code
R1	Allows interaction through the system with console and GUI
R2	Allows any combination of checks to be run on code
R3	Provides functionality to select specific packages to lint
R4	Provides functionality to import specific GitHub files to lint
R5	Allows addition/removal of classes from the selected list to be linted
R6	Outputs readable and informative information after running a set of lint checks
R7	Ensures most of the system is unaffected by changes to imported library ASMReader
R7.1	Combines the system's internal representation of MyLocalVariableNode and MyFieldNode
R8	Allows easy use by any user somewhat familiar with the functionality of a linter
R9	Ability to change the java bytecode of a set of files on a failed lint check
R10	Compiles a file containing the output from the most recent linter check

Software and Architecture Design Specification (SADS):

The architecture we utilize throughout this project is the 3-Layer architecture, in which the layers consist of presentation, domain, and data source. This top-down architecture allows the presentation layer to reference any information from layers below it, in order to handle user input. The domain class can reference the ASM implementation in the data source in order to handle the actual internal functionality of the system, and the data source handles specific data objects alongside outputting information from the system.



Test Plan/Strategy:

Our intent with testing this system was to write as many automated tests as necessary to fully cover both main and edge cases of the system, with some manual testing to ensure the functionality of the GUI elements, and the file loading/storing. We ended up writing a total of 191 tests to ensure this was done to the best of our ability.

We began by writing unit tests for each testable component in the domain, utilizing the EasyMock tool and dependency injection to be able to test each class independently. After those were complete, we wrote integration tests based on the previous unit tests, removing the mocks and replacing them with the other objects themselves. We wrote more tests in these cases to cover more situations necessary to fully integrate related components. Finally, we wrote system/acceptance tests that run through the console UI, and test based on expected text outputs. This makes these tests quite finicky, and unstable. Also outlined below are the manual test inputs and outputs that we used to make sure the system was working properly.

Test Cases:

Automated:

We have comprehensive unit and integration tests written for the following classes:

- ASMParser
- MyClassNode
- MyFieldInsnNode
- MyFieldNode
- MyLocalVariableNode
- MyMethodInsnNode
- MyMethodNode
- MyVarInsnNode
- Sanitizer
- AdapterPattern
- Composition
- FacadePattern
- HollywoodCheck
- MethodLength
- NamesCheck
- RedundantInterfaceCheck
- UnusedInstantiationCheck
- VarStates
- FieldStates

Utilizing the Jacoco code coverage tool, we ensured that the tests we wrote for these classes are comprehensive and hit every line of code written. For this portion of our project, our overall code coverage is at 99%.

The rest of the project is tested through our system/acceptance tests, as it all has to do with the way we handle GUI input/output, and how we actually parse specific bytecode. We ran into a large issue with Gradle while trying to get an accurate code coverage estimate from these tests, as they pass when run through JUnit, but not through Gradle. Therefore, all code coverage estimates done through the project must be done with these tests commented out. However, based on the way we wrote the tests, they should cover most, if not all of the functionality in the following classes (alongside all those unit tested above):

- ASMReader
- Directory
- FileWriter
- GithubImport
- GithubPathInformation
- Grabber
- javaClassParse
- javaFolderParse
- PackageImport
- PopulateJavaFile
- CheckRunner
- ConsoleManager
- InputManager

To test these classes through the system tests, the following situations were modeled:

- Running Name Check on “example” package
- Running Unused Instantiation Check on “example” package
- Running Method Length Check on “example” package
- Running Redundant Interfaces Check on “example” package
- Running Composition Check on “example” package
- Running Hollywood Check on “example” package
- Running Facade Pattern Check on “example.goodFacade” package, expecting detection
- Running Facade Pattern Check on “example.badFacade” package, expecting no detection
- Running Strategy Pattern Check on “example.goodStrategy” package, expecting detection
- Running Adapter Pattern Check on “example.goodAdapter” package, expecting detection
- Running Adapter Pattern Check on “example.goodFacade” package, expecting no detection
- Running the “help” command

- Running All Checks on “example” package
- Removing a class from the list of tested classes from “expected”, then running Names check
- Removing a class from the list of tested classes from “expected”, re-adding that same class, then running Names check
- Retrieving a file through the Github import functionality, then running all checks on it

Manual:

The only part of the system not tested automatically is the GUI. This has all the same functionality as the console, so we were not incredibly concerned with extensively testing this portion. However, we did have a few manual test plans we used to make sure it was working. The steps are described below, for a guide of images to see where the inputs and outputs below should be put is shown in the User Guide above. The different manual tests are as follows:

- Simple Package Test:
    - Begin application, type GUI into the console to select GUI usage
    - Click “Packages”
    - Type in “example”
    - Click “Submit”
    - From the next page, click “Submit” again
    - Select all checks
    - Click “Submit”
    - Expect this as the output:

```
Running checks:
        Adapter Pattern
        Composition Over Inheritance
        Facade Pattern
        Hollywood Principle
        Method Length
        Name Style
        Redundant Interfaces
        Strategy Pattern
        Unused Instantiation

On classes:
        badClass$
        superBadClass$


Composition Over Inheritance:
        Class badClass$ inherits from user created class superBadClass$. Could composition be used instead?

Hollywood Principle Violations:
        Class badClass$ uses field noString from superBadClass$ in method BadMethodName
        Class badClass$ calls method doNothing from superBadClass$ in method BadMethodName
Method Length Check:
        Class: badClass$
                Method: longMethod
                        Method too long: (56 lines) Shorten it to 35 lines or less.
Names Check:
```

```
        Class: badClass$
                Name Style Violations:
                        Class Name checks:
                                Class Name does not start with a capital letter 0987409874
                                Class Name contains the non-alphanumeric character: $
                        Field Name checks:
                                Field string has the same name as its type
                                Field NotGood has an uppercase first letter, and is not static and final
                                Field j has too short of a name (1 character)
                        Method & Method Variable Name checks:
                                Variable Ok has an uppercase first letter in method <init>
                                Variable badclass$ has the same name as its type in method BadMethodName
                                Variable Integer has an uppercase first letter in method BadMethodName
                                Variable i has the same name as its type in method BadMethodName
                                Method BadMethodName has an uppercase first letter
                                Variable myfirstLinter has the same name as its type in method m
                                Method m has too short of a name (1 character)
                                Variable i has the same name as its type in method methodWithUnusedVariables
                                Variable i has the same name as its type in method longMethod
        Class: superBadClass$
                Name Style Violations:
                        Class Name checks:
                                Class Name does not start with a capital letter
                                Class Name contains the non-alphanumeric character: $
Unused Instantiation Check:
        Class: badClass$
                Unused Variables:
                        Line 8: Unused field named string
                        Line 9: Unused field named okay
                        Line 10: Unused field named NotGood
                        Line 12: Unused field named j
                        Line 13: Unused field named unusedString
                        Line 24: Unused field named noString
                        Line 26: Unused field named okay
                        Line 20: Unused variable named Integer in method BadMethodName
                        Line 30: Unused variable named myfirstLinter in method m
                        Line 34: Unused variable named name in method methodWithUnusedVariables
                        Line 35: Unused variable named number in method methodWithUnusedVariables
                        Line 36: Unused variable named newNumber in method methodWithUnusedVariables
                        Line 41: Unused variable named newNumber in method methodWithUnusedVariables
                        Line 50: Unused variable named newNumber in method methodWithUnusedVariables
                        Line 51: Unused variable in method methodWithUnusedVariables
        Class: superBadClass$
                Unused Variables:
                        Line 4: Unused field named noString
                        Line 5: Unused field named okay
```

- Complex Package Test:
    - Begin application, type GUI into the console to select GUI usage
    - Click “Packages”
    - Type in “example”
    - Click “Submit”
    - Type “example.badClass$” into the text box
    - Click “Remove”
    - Click “Submit”
    - Select “Method Length,” Names,” and “Unused Instantiation” checks
    - Click “Submit”
    - Expect this as the output:

```
Running checks:
        Method Length
        Name Style
        Unused Instantiation

On classes:
        superBadClass$

Names Check:
        Class: superBadClass$
                Name Style Violations:
                        Class Name checks:
                                Class Name does not start with a capital letter
                                Class Name contains the non-alphanumeric character: $
Unused Instantiation Check:
        Class: superBadClass$
                Unused Variables:
                        Line 4: Unused field named noString
                        Line 5: Unused field named okay
```

- Github Test:
    - Begin application, type GUI into the console to select GUI usage
    - Click “Github”
    - Type in “https://github.com/TheAlgorithms/Java/tree/master/src/main/java/com/thealgorithms/backtracking”
    - Click “Submit”
    - Type “data_source.KnightsTour ” into the text box
    - Click “Remove”
    - Type “data_source.FloodFill” into the text box
    - Click “Remove”
    - Type “data_source.KnightsTour ” into the text box
    - Click “Add”
    - Click “Submit”
    - Select all checks
    - Click “Submit”
    - Expect this as the output:

```
Running checks:
        Adapter Pattern
        Composition Over Inheritance
        Facade Pattern
        Hollywood Principle
        Method Length
        Name Style
        Redundant Interfaces
        Strategy Pattern
        Unused Instantiation

On classes:
```

```
        NQueens
        Permutation
        PowerSum
        Combination
        KnightsTour

Names Check:
        Class: PowerSum
                Name Style Violations:
                        Method & Method Variable Name checks:
                                Method Sum has an uppercase first letter
```